ADVANCED SCIENTIFIC LIBRARY
ASL C INTERFACE
User's Guide
<Basic Functions Vol.4>

PROPRIETARY NOTICE

PREFACE

This manual describes general concepts, functions, and specifications for use of the Advanced Scientific Library (ASL) C interface.

The manuals corresponding to this product consist of seven volumes, which are divided into the chapters shown below. This manual describes the basic functions, volume 4.

Basic Functions Volume 1

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Storage Mode Conversion	Explanation of algorithms, method of using, and usage example of function related to storage mode conversion of array data.
3	Basic Matrix Algebra	Explanation of algorithms, method of using, and usage example of function related to basic calculations involving matrices.
4	Eigenvalues and Eigenvectors	Explanation of algorithms, method of using, and usage example of function related to **the standard eigenvalue problem** for real matrices, complex matrices, real symmetric matrices, Hermitian matrices, real symmetric band matrices, real symmetric tridiagonal matrices, real symmetric random sparse matrices, Hermitian random sparse matrices and **the generalized eigenvalue problem** for real matrices, real symmetric matrices, Hermitian matrices, real symmetric band matrices.

Basic Functions Volume 2

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Simultaneous Linear Equations (Direct Method)	Explanation of algorithms, method of using, and usage example of function related to simultaneous linear equations corresponding to real matrices, complex matrices, positive symmetric matrices, real symmetric matrices, Hermitian matrices, real band matrices, positive symmetric band matrices, real tridiagonal matrices, real upper triangular matrices, and real lower triangular matrices.

Basic Functions Volume 3

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Fourier Transforms and their applications	Explanation of algorithms, method of using, and usage example of function related to one-, two- and three-dimensional complex Fourier transforms and real Fourier transforms, one-, two- and three-dimensional convolutions, correlations, and power spectrum analysis, wavelet transforms, and inverse Laplace transforms.

Basic Functions Volume 4

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Differential Equations and Their Applications	Explanation of algorithms, method of using, and usage example of function related to **ordinary differential equations initial value problems** for high-order simultaneous ordinary differential equations, implicit simultaneous ordinary differential equations, matrix type ordinary differential equations, stiff problem high-order simultaneous ordinary differential equations, simultaneous ordinary differential equations, first-order simultaneous ordinary differential equations, and high-order ordinary differential equations, and **ordinary differential equations boundary value problems** for high-order simultaneous ordinary differential equations, first-order simultaneous ordinary differential equations, high-order ordinary differential equations, high-order linear ordinary differential equations, and second-order linear ordinary differential equations, and **integral equations** for Fredholm's integral equations of second kind and Volterra's integral equations of first kind, and **partial differential equations** for two- and three-dimensional inhomogeneous Helmholtz equation.
3	Numerical Differentials	Explanation of algorithms, method of using, and usage example of function related to numerical differentials of one-variable functions and multi-variable functions.
4	Numerical Integration	Explanation of algorithms, method of using, and usage example of function related to numerical integration over a finite interval, semi-infinite interval, fully infinite interval, two-dimensional finite interval, and multi-dimensional finite interval.
5	Interpolations and Approximations	Explanation of algorithms, method of using, and usage example of function related to interpolations, surface interpolations, least squares approximations, least squares surface approximations, and Chebyshev's approximations.
6	Spline Functions	Explanation of algorithms, method of using, and usage example of function related to interpolation, smoothing, numerical derivatives, and numerical integrals using cubic splines, bicubic splines and B–splines.

Basic Functions Volume 5

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Special Functions	Explanation of algorithms, method of using, and usage example of function related to Bessel functions, modified Bessel functions, spherical Bessel functions, functions related to Bessel functions, Gamma functions, functions related to Gamma functions, elliptic functions, indefinite integrals of elementary functions, associated Legendre functions, orthogonal polynomials, and other special functions.
3	Sorting and Ranking	Explanation and usage examples of function related to sorting and ranking.
4	Roots of Equations	Explanation of algorithms, method of using, and usage example of function related to roots of algebraic equations, nonlinear equations, and simultaneous nonlinear equations.
5	Extremal Problems and Optimization	Explanation of algorithms, method of using, and usage example of function related to minimization of functions with no constraints, minimization of the sum of the squares of functions with no constraints, minimization of one-variable functions with constraints, minimization of multi-variable functions with constraints, and shortest path problem.

Basic Functions Volume 6

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Random Number Tests	Explanation and usage examples of function related to uniform random number tests, and distribution random number tests.
3	Probability Distributions	Explanation and usage examples of function related to continuous distributions and discrete distributions.
4	Basic Statistics	Explanation and usage examples of function related to basic statistics, variance-covariance and correlation.
5	Tests and Estimates	Explanation and usage examples of function related to interval estimates and tests.
6	Analysis of Variance and Design of Experiments	Explanation and usage examples of function related to one-way layout, two-way layout, multiple-way layout, randomized block design, Greco-Latin square method, cumulative Method.
7	Nonparametric Tests	Explanation and usage examples of function related to tests using χ^2 distribution and tests using other distributions.
8	Multivariate Analysis	Explanation and usage examples of function related to principal component analysis, factor analysis, canonical correlation analysis, discriminant analysis, cluster analysis.
9	Time Series Analysis	Explanation and usage examples of function related to autocorrelation, cross correlation, autocovariance, cross covariance, smoothing and demand forecasting.
10	Regression analysis	Explanation and usage examples of function related to linear Regression and nonlinear Regression.

Shared Memory Parallel Functions

Chapter	Title	Contents
1	Introduction	Explanation of the organization of this manual, how to view each item, and usage limitations.
2	Basic Matrix Algebra	Explanation of algorithms, method of using, and usage example of function related to obtain the product of real matrices and complex matrices.
3	Simultaneous Linear Equations (Direct Method)	Explanation of algorithms, method of using, and usage example of function related to simultaneous linear equations corresponding to real matrices, complex matrices, real symmetric matrices, and Hermitian matrices.
4	Simultaneous Linear Equations (Iteration Method)	Explanation of algorithms, method of using, and usage example of function related to simultaneous linear equations corresponding to real positive definite symmetric sparse matrices, real symmetric sparse matrices and real asymmetric sparse matrices.
5	Eigenvalues and Eigenvectors	Explanation of algorithms, method of using, and usage example of function related to the eigenvalue problem for real symmetric matrices and Hermitian matrices.
6	Fourier Transforms and their applications	Explanation of algorithms, method of using, and usage example of function related to one-, two- and three-dimensional complex Fourier transforms and real Fourier transforms, two- and three-dimensional convolutions, correlations, and power spectrum analysis.
7	Sorting	Explanation and usage examples of function related to sorting and ranking.

Document Version 3.0.0-230301 for ASL, March 2023

Remarks

(1) This manual corresponds to ASL 1.1. All functions described in this manual are program products.

(2) Proper nouns such as product names are registered trademarks or trademarks of individual manufacturers.

(3) This library was developed by incorporating the latest numerical computational techniques. Therefore, to keep up with the latest techniques, if a newly added or improved function includes the function of an existing function may be removed.

Contents

Chapter 1

INTRODUCTION

1.1 OVERVIEW

1.1.1 Introduction to The Advanced Scientific Library ASL C interface

Table 1−1 lists correspondences among product categories, functions of ASL and supported hardware platforms. Interfaces of those functions that have the same name and that belong to the same version of ASL are common among hardware platforms.

Table 1−1 Classification of functions included in ASL

Classification of Functions	Volume
Basic functions	Vol. 1-6
Shared memory parallel functions	Vol. 7

1.1.2 Distinctive Characteristics of ASL C interface

ASL C interface has the following distinctive characteristics.

(1) Functions are optimized using compiler optimization to take advantage of corresponding system hardware features.

(2) Special-purpose functions for handling matrices are provided so that the optimum processing can be performed according to the type of matrix (symmetric matrix, Hermitian matrix, or the like). Generally, processing performance can be increased and the amount of required memory can be conserved by using the special-purpose functions.

(3) Functions are modularized according to processing procedures to improve reliability of each component function as well as the reliability and efficiency of the entire system.

(4) Error information is easy to access after a function has been used since error indicator numbers have been systematically determined.

1.2 KINDS OF LIBRARIES

Numeric storage units of ASL C interface is 4-byte.

Table 1−2 Kinds of libraries providing ASL C interface

Size of variable(byte)		Declaration of arguments	Kind	Kind of library
integer	real			
4	8	int double	32bit integer Double-precision function	32bit integer library (link option: -lasl_sequential)
4	4	int float	32bit integer Single-precision function	
8	8	long double	64bit integer Double-precision function	64bit integer library (link option: -lasl_sequential_i64)
8	4	long float	64bit integer Single-precision function	

(∗1) Functions that appear in this documentation do not always support all of the four kinds of functions listed above. For those functions that do not support some of those function kinds, relevant notes will appear in the corresponding subsections.

(∗2) For compiling the program with functions in the 64-bit integer library, the option "`-DASL_LIB_INT64`" must be specified (See the Note (2) in 1.5).

1.3 ORGANIZATION

This section describes the organization of Chapters 2 and later.

1.3.1 Introduction

The first section of each chapter is a general introduction describing such information as the effective ways of using the functions, techniques employed, algorithms on which the functions are based, and notes.

1.3.2 Organization of Function Description

The second section of each chapter sequentially describes the following topics for each function.

(1) Function

(2) Usage

(3) Arguments and return value

(4) Restrictions

(5) Error indicator (Return Value)

(6) Notes

(7) Example

Each item is described according to the following principles.

1.3.3 Contents of Each Item

(1) **Function**
Function briefly describes the purpose of the ASL C interface function.

(2) **Usage**
Usage describes the function name and the order of its arguments. In general, arguments are arranged as follows. When an argument is an address-passing variable, & is appended in front of the argument name.

ierr = function-name (input-arguments, input/output-arguments, output-arguments, isw, work);

isw is an input argument for specifying the processing procedure. ierr is a return value. In some cases, input/output arguments precede input arguments. The following general principles also apply.

- Array are placed as far to the left as possible according to their importance.
- The dimension of an array immediately follows the array name. If multiple arrays have the same dimension, the dimension is assigned as an argument of only the first array name. It is not assigned as an argument of subsequent array names.

(3) **Arguments and return value**
Arguments and return value are explained in the order described above in paragraph (2). The explanation format is as follows.

Arguments and return value	Type	Size	Input/Output	Contents
(a)	(b)	(c)	(d)	(e)

(a) Arguments and return value

Arguments and return value are explained in the order they are designated in the Usage paragraph.

(b) Type

Type indicates the data type of the argument. Any of the following codes may appear as the type.

I : Integer type

D : Double precision real

R : Real

Z : Double precision complex

C : Complex

There are 64-bit integer and 32-bit integer for integer type arguments. In a 32-bit (64-bit) integer type function, all the integer type arguments are 32-bit (64-bit) integer. In other words, kinds of libraries determine the sizes of integer type arguments (Refer to 1.4). In the user program, a 32-bit/64-bit integer type argument must be declared by `int`/ `long`, respectively.

(c) Size

Size indicates the required size of the specified argument. If the size is greater than 1, the required area must be reserved in the program calling this function.

1 : Indicates that argument is a variable.

n : Indicates that the argument is a vector (one-dimensional array) having n elements. The argument n indicating the size of this vector is defined immediately after the specified vector. However, if the size of a vector or array defined earlier, it is omitted following subsequently defined vectors or arrays. The size may be specified by only a numeric value or in the form of a product or sum such as $3 \times n$ or $n + m$.

(d) Input/Output

Input/Output indicates whether the explanation of argument contents applies to input time or output time.

i. When only "Input" appears

When the control returns to the program using this function, information when the argument is input is preserved. The user must assign input-time information unless specifically instructed otherwise. When the argument is a variable, the variable value must be passed.

ii. When only "Output" appears

Results calculated within the function are output to the argument. No data is entered at input time. When the argument is a variable, the variable address must be passed.

iii. When both "Input" and "Output" appear

Argument contents change between the time control passes to the function and the time control returns from the function. The user must assign input-time information unless specifically instructed otherwise. When the argument is a variable, the variable address must be passed.

iv. When "Work" appears

Work indicates that the argument is an area used when performing calculations within the function. A work area having the specified size must be reserved in the program calling this function. The contents of the work area may have to be maintained so they can be passed along to the next calculation.

(e) Contents

Contents describes information held by the argument at input time or output time.

- A sample Argument description follows.

Example

The statement of the function (ASL_dbgmlc, ASL_rbgmlc) that obtains the LU decomposition and the condition number of a real matrix is as follows.

Double precision:

```
ierr = ASL_dbgmlc  (a, lna, n, ipvt, &cond, w1);
```

Single precision:

```
ierr = ASL_rbgmlc  (a, lna, n, ipvt, &cond, w1);
```

The explanation of the arguments and return value is as follows.

Table 1−3 Sample Arguments and Return Value

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	a	Note {D∗ / R∗}	lna×n	Input	Real matrix A(two-dimensional array)
				Output	The matrix A decomposed into the matrix LU where U is a unit upper triangular matrix and L is a lower triangular matrix.
2	lna	I	1	Input	Adjustable dimension size of array a
3	n	I	1	Input	Order n of matrix A
4	ipvt	I*	n	Output	Pivoting information ipvt[i−1]: Number of the row exchanged with row i in the i-th step.
5	cond	{D / R}	1	Output	Reciprocal of the condition number
6	w1	{D∗ / R∗}	n	Work	Work area
7	ierr	I	1	Output	Error indicator (Return Value)

To use this function, arrays a, ipvt and w1 must first be allocated in the calling program so they can be used as arguments. a is a {double-precision / single-precision}Note real array of size [lna × n] , ipvt is an integer array of size n and w1 is a {double-precision / single-precision}real array of size n.

When the 64-bit integer version is used, all integer-type arguments (lna, n, ipvt and ierr) must be declared by using `long`, not `int`.

Note The entries enclosed in brace { } mean that the array should be declared double precision type when using function ASL_dbgmlc and real type when using function ASL_rbgmlc. Braces are used in this manner throughout the remainder of the text unless specifically stated otherwise.

Data must be stored in a, lna and n before this function is called. The LU decomposition and condition number of the assigned matrix are calculated with in the function, and the results are stored in array a and variable cond. In addition, pivoting information is stored in ipvt for use by subsequent functions.

ierr is a return value used to notify the user of invalid input data or an error that may occur during processing. If processing terminates normally, ierr is set to zero.

Since w1 is a work area used only within the function, its contents at input and output time have no special meaning.

(4) **Restrictions**

Restrictions indicate limiting ranges for function arguments.

(5) **Error indicator (Return Value)**

Each function has been given an error indicator as a return value. This error indicator, which has uniformly been given the variable name ierr, is placed at the end of the arguments. If an error is detected within the function, a corresponding value is output to ierr. Error indicator values are divided into five levels.

Table 1−4 Classification of Return Values

Level	Return value	Meaning	Processing result
Normal	0	Processing is terminated normally.	Results are guaranteed.
Warning	1000∼2999	Processing is terminated under certain conditions.	Results are conditionally guaranteed.
Fatal	3000∼3499	Processing is aborted since an argument violated its restrictions.	Results are not guaranteed.
	3500∼3999	Obtained results did not satisfy a certain condition.	Obtained results are returned (the results are not guaranteed).
	4000 or more	A fatal error was detected during processing. Usually, processing is aborted.	Results are not guaranteed.

(6) **Notes**

Notes describes ambiguous items and points requiring special attention when using the function.

(7) **Example**

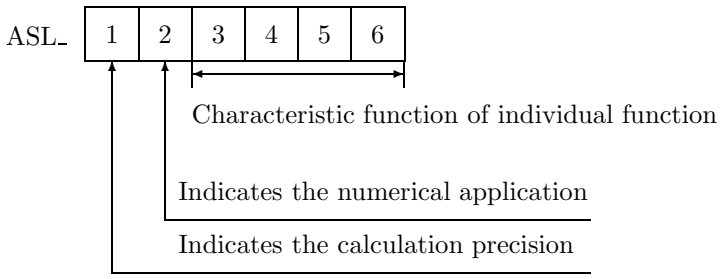
Here gives an example of how to use the function. Note that in some cases, multiple functions are combined in a single example. The output results are given in the 32-bit integer version, and may differ within the range of rounding error if the compiler or intrinsic functions are different.

In addition, when the 64-bit integer version library is used, the `long`-type conversion specification to be given to `printf` or `scanf` must be `%ld`. The source codes of examples in this document are included in User's Guide. Input data, if required, is also included in it. To build up an executable files by compiling these example source codes, they should be linked with this product library.

1.4 FUNCTION NAMES

The functions name of ASL C interface basic functions consists of ten characters with a prefix "ASL_" and ⟨six alphanumeric characters⟩.

Figure 1−1 Function Name Components



"1" in Figure 1−1 : The following eight letters are used to indicate the calculation precision.

d, w	Double precision real-type calculation
r, v	Single precision real-type calculation
z, j	Double precision complex-type calculation
c, i	Single precision complex-type calculation

However, the complex type calculations listed above do not necessarily require complex arguments.

"2" in Figure 1−1 : Currently, the following letters letterererererere are used to indicate the application field in the ASL C interface related products.

Letter	Application Field	Volume
a	Storage mode conversion	1
	Basic matrix algebra	1, 7
b	Simultaneous linear equations (direct method)	2, 7
c	Eigenvalues and eigenvectors	1, 7
f	Fourier transforms and their applications	3, 7
	Time series analysis	6
g	Spline function	4
h	Numeric integration	4
i	Special function	5
j	Random number tests	6
k	Ordinary differential equation (initial value problems)	4
l	Roots of equations	5
m	Extremum problems and optimization	5
n	Approximation and regression analysis	4, 6
o	Ordinary differential equations (boundary value problems), integral equations and partial differential equations	4
p	Interpolation	4
q	Numerical differentials	4

Letter	Application Field	Volume
s	Sorting and ranking	5, 7
x	Basic matrix algebra	1
	Simultaneous linear equations (iterative method)	7
1	Probability distributions	6
2	Basic statics	6
3	Tests and estimates	6
4	Analysis of variance and design of experiments	6
5	Nonparametric tests	6
6	Multivariate analysis	6

"3–6" in Figure 1−1 **:** These characters indicate the characteristic function of the individual function.

1.5 NOTES

(1) To use ASL C interface, the header file `asl.h`must be included.

(2) For compiling the program with functions in ASL C interface 64-bit integer library, the compile option "`-DASL_LIB_INT64`" must be specified. This option will activate the prototype declaration for 64-bit integer functions in the header file `asl.h`, and without the option "`-DASL_LIB_INT64`", those for 32-bit integer functions will be activated.

(3) The name "⟨6 lowercase letters⟩ following ASL_ " is reserved by ASL C interface.

(4) For using 64-bit integer library, you must use "`long`" for integer type declaration. Otherwise, use "`int`" for integer type declaration.

(5) Use the functions of double precision version whenever possible. They not only provide higher precision solutions but also are more stable than single precision versions, in particular, for eigenvalue and eigenvector problems.

(6) To suppress compiler operation exceptions, ASL C interface functions are set to so that they conform to the compiler parameter indications of a user's main program. Therefore, the main program must suppress any operation exceptions.

(7) The numerical calculation programs generally deal with operations on finite numbers of digits, so the precision of the results cannot exceed the number of operation digits being handled. For example, since the number of operation digits (in the mantissa part) for double-precision operations is on the order of 15 decimal digits, when using these floating point modes to calculate a value that mathematically becomes 1, an error on the order of 10^{-15} may be introduced at any time. Of course, if multiple length arithmetic is emulated such as when performing operations on an arbitrary number of digits, this kind of error can be controlled. However, in this case, when constants such as π or function approximation constants, which are fixed in double-precision operations, for example, are also to be subject to calculations that depend on the length of the multiple length arithmetic operations, the calculation efficiency will be worse than for normal operations.

(8) A solution cannot be obtained for a problem for which no solution exists mathematically. For example, a solution of simultaneous linear equations having a singular (or nearly singular) matrix for its coefficient matrix theoretically cannot be obtained with good precision mathematically. Numerical calculations cannot strictly distinguish between mathematically singular and nearly singular matrices. Of course, it is always possible to consider a matrix to be singular if the calculation value for the condition number is greater than or equal to an established criterion value.

(9) Generally, if data is assigned that causes a floating point exception during calculations (such as a floating point overflow), a normal calculation result cannot be expected. However, a floating point underflow that occurs when adding residuals in an iterative calculation is an exception to this.

(10) For problems that are handled using numerical calculations (specifically, problems that use iterative techniques as the calculation method), there are cases in which a solution cannot be obtained with good precision and cases in which no solution can be obtained at all, by a special-purpose function.

(11) Depending on the problem being dealt with, there may be cases when there are multiple solutions, and the execution result differs in appearance according to the compiler used or the computer or OS under which

the program is executed. For example, when an eigenvalue problem is solved, the eigenvectors that are obtained may differ in appearance in this way.

(12) The mark "DEPRECATED" denotes that the subroutine will be removed in the future. Use **ASL Unified Interface**, the higher performance alternative practice instead.

Chapter 2

DIFFERENTIAL EQUATIONS AND THEIR APPLICATIONS

2.1 INTRODUCTION

This chapter describes functions for ordinary differential equation initial value problems and boundary value problems and integral equations and partial differential equations. Initial value problem functions sequentially obtain approximate solutions at discrete points from ordinary differential equations, Ordinary differential equation (boundary value problem) and partial differential equation functions obtain approximate solutions at arbitrary points within the boundary, and integral equation functions obtain approximate solutions at arbitrary points.
Since this library functions described in Sections 2.2.1 through 2.2.4 and Sections 2.3.1 through 2.3.2 correspond to high-order simultaneous ordinary differential equations, they can be used directly, without having to replace the equations by simultaneous ordinary differential equations of the 1st order. Also, simultaneous equations of the 1st order or independent equations of the 1st order can be solved by letting the argument for the order or the argument for the number of simultaneous equations be 1.
If you want to obtain the solution orbit when the step size is automatically controlled in the initial value problem, you also can output the solution for each step size besides outputting the solution at the desired point.
In the boundary value problem, you can use both numerical boundary conditions function, which determine boundary conditions by user specified function values and derivative values at boundary, and function boundary conditions function, which determine boundary conditions by user specified functions and derivatives at boundary.
This library provides functions corresponding to equations having the following characteristics.

Ordinary Differential Equations (Initial Value Problems)

(1) High-order simultaneous ordinary differential equations (speed priority), simultaneous ordinary differential equations of the 1st order, and high-order ordinary differential equations
Obtains solutions by using the Runge-Kutta-Verner method with automatic step-size control. Most efficient for non-stiff and weakly stiff problems when the function evaluation cost is low or the precision requirements are not high.

(2) High-order simultaneous ordinary differential equations (precision priority)
Obtains solutions by using a linear multistep method with automatic step-size and automatic degree control. Most efficient for non-stiff and weakly stiff problems when the function evaluation cost is high or the precision requirements are high.

(3) Implicit simultaneous ordinary differential equations
Implicit ordinary differential equations have the form:

$$f_i(x, y_1, y_2, \cdots, y_n, y_1', y_2', \cdots, y_n', \cdots) = 0 \quad \text{for} \quad (i = 1, 2, \cdots, n)$$

where $\dfrac{\partial f_i}{\partial y_j}$ for $(i = 1, 2, \cdots, n; j = 1, 2, \cdots, n)$ are non-singular equations. For example, this corresponds to equations such as the following in which y_1', (or y_2') depends on both equation (1) and

equation (2) and cannot be determined from the evaluation of an independent equation.

$$\begin{cases} y_1'y_2' - x^2 = 0 & \text{Initial value } y_1(x_0) = y_{10} \cdots (1) \\ y_1' + y_2' - y_1 - 2x = 0 & \text{Initial value } y_2(x_0) = y_{20} \ldots (2) \end{cases}$$

In addition, this function can solve ordinary differential equations that are taken to be simultaneous with algebraic equations.

An example of ordinary differential equations that are taken to be simultaneous with algebraic equations is as follows. It consists of one group of algebraic equations or nonlinear equations and one group of ordinary differential equations.

$$\begin{cases} y_1'y_2' - x^2 = 0 & \text{Initial value } y_1(x_0) = y_{10} \cdots (1) \\ y_2' + y_3 + 2x = 0 & \text{Initial value } y_2(x_0) = y_{20} \ldots (2) \\ 2y_1 + 3y_2 + y_3 - 1 = 0 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3) \end{cases}$$

This kind of problem can be solved only by using this function.

Although this function also can solve general high-order simultaneous ordinary differential equations, the calculation efficiency is poor compared with other functions since the integration is carried out while solving the nonlinear equations. This function also can solve nonlinear simultaneous equations or nonlinear equations. In this case, partial differential coefficients or differential coefficients need not be input.

(4) Stiff problem high-order simultaneous ordinary differential equations

A stiff problem is a problem for which the solution consists of two or more factors that vary with different scales for variations of the independent variable. An example of a stiff problem is to solve the differential equation $y'' = \lambda y \ \ (\lambda \gg 0)$ with initial conditions $(y = 1, y'' = -\lambda \ \text{ at } \ x = 0)$. The general solution of this differential equation is given by a linear combination of the factors $e^{\lambda x}$ and $e^{-\lambda x}$, which vary with different scales for variations in the independent variable x. Although the correct solution of this problem is $y = e^{-\lambda x}$, if the solution at $x = x_1$ is sought by using:

$$y = e^{-\lambda x_1} + \varepsilon = e^{-\lambda x_1} + \varepsilon' e^{\lambda x_1} \quad (\varepsilon, \varepsilon' : \text{Error in numerical calculations})$$

then as x becomes large:

$$y \to \varepsilon' e^{\lambda x}$$

which has the property that it separates from the correct solution.

This function can be used to efficiently solve such strongly stiff problems.

(5) Ordinary differential equation of the type $M\boldsymbol{y''} + C\boldsymbol{y'} + K\boldsymbol{y} = \boldsymbol{p}(x)$

This function solves the ordinary differential equation $M\boldsymbol{y''} + C\boldsymbol{y'} + K\boldsymbol{y} = \boldsymbol{p}(x)$, which is known as the equation of motion. M, C and K are $n \times n$ matrices called the mass matrix, damping matrix, and stiffness matrix respectively, and $\boldsymbol{p}(x)$ is the external force vector. n represents the number of simultaneous equations. This function carries out processing so that a solution can be obtained even if the mass matrix M is singular such as if it includes zero for a diagonal term (there is a point having mass zero). Also, solutions are obtained for each entered step-size.

Ordinary Differential Equations (Boundary Value Problems)

(1) High-order simultaneous ordinary differential equations, first-order simultaneous ordinary differential equations, and high-order ordinary differential equations

Solves the boundary value problem by finding a suitable initial value according to the multipoint shooting method based on the Runge-Kutta-Verner method. Automatically sets shooting points so

that solutions are obtained accurately and more efficiently than by the general multipoint shooting method and parameterizes the nonlinear calculation portion.

(2) High-order linear ordinary differential equations
Solves the boundary value problem by combining the collocation method within the weighted residual method and the B-spline function. If the ordinary differential equations are linear, solutions can be obtained quickly and precisely.

(3) Second-order linear ordinary differential equations
Generalizes the coefficient determination method for second order ordinary differential equations. Effective for second-order linear ordinary differential equations.

Integral Equations

(1) Fredholm's integral equation of the second kind
Gauss' integration method is used to solve the integral equation to obtain a solution at an arbitrary point by interpolating using a cubic spline function.

(2) Volterra's integral equation of the first kind
Maclaurin's formula is used to solve the integral equation to obtain a solution at an arbitrary point by interpolating using a cubic spline function.

Partial Differential Equations

(1) Two-dimensional inhomogeneous Helmholtz equation
This function uses a two-dimensional five-point difference in a given rectangular area to solve the inhomogeneous Helmholtz equation.

(2) Three-dimensional inhomogeneous Helmholtz equation
This function uses a three-dimensional seven-point difference in a given rectangular area to solve the inhomogeneous Helmholtz equation.

2.1.1 Notes

2.1.1.1 Ordinary Differential Equations (Initial Value Problems)

(1) Since none of these functions can obtain the correct solution if a point of discontinuity is included in the calculation range, the calculation range must be divided at any point of discontinuity that exists, and separate calculations must be performed for each subdivided range.

(2) The step size must be extremely small for functions having severe oscillations. However, if the interval for which the problem is to be solved increases, then the cumulative error will increase and precision will decrease.

(3) In many cases, it is unclear whether or not the equations are stiff. In such cases, first try solving the equations by using the function 2.2.2 $\begin{Bmatrix}\text{ASL\_dksnca}\\ \text{ASL\_rksnca}\end{Bmatrix}$. If ierr = 4000 is output by that function, then the equations are considered to be stiff and the function 2.2.4 $\begin{Bmatrix}\text{ASL\_dkssca}\\ \text{ASL\_rkssca}\end{Bmatrix}$ should be used to solve them.

(4) The functions 2.2.1 $\begin{Bmatrix}\text{ASL\_dksncs}\\ \text{ASL\_rksncs}\end{Bmatrix}$, 2.2.2 $\begin{Bmatrix}\text{ASL\_dksnca}\\ \text{ASL\_rksnca}\end{Bmatrix}$, and 2.2.4 $\begin{Bmatrix}\text{ASL\_dkssca}\\ \text{ASL\_rkssca}\end{Bmatrix}$ are used for high-order simultaneous ordinary differential equations. However, if the equations are given as conventional simultaneous ordinary differential equations of the 1st order, then do the following.

Conventional	
Function f(x, y, yp)	$\begin{cases}\text{yp[0]} = f_1(\text{x}, \text{y[0]}, \cdots, \text{y[n}-1\text{]});\\ \quad\vdots\\ \text{yp[n}-1\text{]} = f_n(\text{x}, \text{y[0]}, \cdots, \text{y[n}-1\text{]});\end{cases}$
Arguments	Interval $x \sim x_f$: x, xf Initial value ${y_i}^0$: y[0], y[1], ···, y[n-1] Number of simultaneous equations n: n

These functions	
Function f(x, y, n)	$\begin{cases}\text{y[n]} = f_1(\text{x}, \text{y[0]}, \cdots, \text{y[n}-1\text{]});\\ \quad\vdots\\ \text{y[2}\times\text{n}-1\text{]} = f_n(\text{x}, \text{y[0]}, \cdots, \text{y[n}-1\text{]});\end{cases}$
Arguments	Interval $x \sim x_f$: x, xf Initial value ${y_i}^0$: y[0], y[1], ···, y[n-1] Number of simultaneous equations n: n
Additional arguments	Value greater the maximum differential order: mx=1 Differential order of each equation: m[i−1]=1 for (i=1, ···, n)

That is, y becomes the array y[n*(mx+1)], and the function and initial values must be assigned with y[$i-1$+n] for $(i = 1, \cdots$,n) for y_i' and y[$i-1$] for $(i = 1, \cdots$,n) for y_i. Note that the arguments of the function for calculating y also differ. mx and m[$i-1$] are also required as additional arguments, and the value 1 must be set for each of them.
If high-order ordinary differential equations are converted to simultaneous type equations, the precision tends to decrease. Therefore, equations should be created directly as high-order equations as much as possible.

(5) Since all functions except function 2.2.7 $\left\{\begin{matrix}\text{ASL\_dkmncn}\\ \text{ASL\_rkmncn}\end{matrix}\right\}$ output y_i through ${y_i}^{(m)}$ at the xf point or for each step size when automatic step size control is performed, if y_i through ${y_i}^{(m)}$ are required at various points, then either the function must be used continuously while varying xf or output for each automatic step size must be specified and the function must be used continuously with xf fixed at the final point. At this time, the output arguments should be set so they become input arguments for the next execution, and none of the other argument values should be changed, except possibly the xf value.

(6) Function 2.2.7 $\left\{\begin{matrix}\text{ASL\_dkmncn}\\ \text{ASL\_rkmncn}\end{matrix}\right\}$ is used differently than the other functions since the function is given in the form of coefficient matrices and values are output for each step size.
The solution at the point x_f where the solution is required is obtained as follows. The point where the initial value is given is assumed to be x, $(x_f - x)$ is divided by an integer k and this value is assumed to be the step size Δx. The values ${y_i}^{(j)}$ for $(i = 0, 1, 2)$ are obtained for ${y_i}^{(j)}$ at the point x for $(i = 1, \cdots,$ number of simultaneous equations; $j = 0, 1)$ as initial values, then ${y_i}^{(j)}$ at the point advanced from the previous point by Δx is obtained using the previously obtained ${y_i}^{(j)}$ as initial values, and this task is repeated k times. The larger the value of k that is used, the higher will be the precision.

(7) The functions f is created as follows.
Example:
Let f has the same name in the all functions

```
/* C interface example for ASL_dk··· */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <asl.h>
```

- Function f

```
void FORTRAN f (···)
{
        ≀
}
```

- Main function

```
int main()
{
        ≀
   ierr = ASL_dk···(f ,···);
```

```
    }
    return 0;
}
```

2.1.1.2 Ordinary Differential Equations (Boundary Value Problems)

(1) Since the correct solution is not obtained if there is a point of discontinuity within the interval, the boundary value problem is solved by subdividing the interval at the point of discontinuity.

(2) The function used for nonlinear equations parameterizes the equations by multiplying the nonlinear part by α.
The nonlinear part consists of a multiplication or division of two or more of the ${y_i}^{(j)}$ (i:Array number, j:Differential order) such as $y_1 y_2$, $\frac{y_3'}{y_4''}$, $y_1 y_1'$ or $(y_1')^2$ or a function other than a sum or scalar multiple of ${y_i}^{(j)}$ such as $\sin(y_1')$ or $\mid y_2 \mid$.
For example, the differential equations:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - y_1 y_2 \end{cases}$$

are parameterized as follows:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - \alpha y_1 y_2 \end{cases}$$

If there is no nonlinear part, it is unnecessary to parameterize the equations by multiplying by α.
Regardless of whether or not there is a nonlinear part, if parameterizing is not performed, the solution method will be the same as the general multipoint shooting method. This increases the amount of calculations, and in some cases, no solution can be found.
If the linear part is parameterized by mistake, the amount of calculations will be increased, but the solution will not be affected.

(3) When assigning boundary conditions by numerical values, distinguish the starting and ending points by in, the element number by ib, the differential order by ic, and the value at that boundary by bn. At this time, the element number of in, ib, ic, and bn must correspond to the respective boundary conditions. The boundary condition input order is arbitrary.

(4) When assigning boundary conditions by a function, if the value of ${y_i}^{(j)}$ at the starting point is ${ya_i}^{(j)}$ and the value at the ending point is ${yb_i}^{(j)}$, determine the boundary conditions by the function $g_k({ya_i}^{(j)}, {yb_i}^{(j)}) = 0$. At this time, the boundary condition input order is arbitrary.

(5) Since no error check of the differential order or number of simultaneous differential equations can be performed within the function, enter these values carefully.

2.1.1.3 Integral Equations

(1) Since neither of these functions can obtain the correct solution if there is a point of discontinuity within the interval, the interval must be subdivided at the point of discontinuity and separate calculations must be performed in each subdivision.

(2) In 2.4.2 $\left\{\begin{matrix}\text{ASL\_doiev1}\\ \text{ASL\_roiev1}\end{matrix}\right\}$, if the number of subdivisions of the integration interval is too large, the cumulative error increases and precision declines.

2.1.2 Algorithms Used

2.1.2.1 Ordinary Differential Equations (Initial Value Problems)

(1) **Runge-Kutta-Verner method**

Since this method enables the truncation error to be estimated, the desired solution is obtained with automatic control of the step size complying with the more lenient value of the required local absolute precision and the required local relative precision.

A single iteration of the Verner method for step size h is expressed as follows for $y' = f(x, y)$.

$$\begin{aligned} y_{n+1} &= y_n + \sum_{i=1}^{8} \gamma_i k_i \\ E &= \sum_{i=1}^{8} \gamma_i^* k_i \\ k_i &= hf\left(x_n + \alpha_i h, y_n + h\sum_{j=1}^{i-1} \beta_{ij} k_j\right) \quad (i = 1, 2, \cdots, 8) \end{aligned}$$

In the expressions shown above, y_{n+1} is the approximate solution having sixth order truncation precision,

Table 2−1 Coefficient Table

i	α_i	β_{i1}	β_{i2}	β_{i3}	β_{i4}	β_{i5}	β_{i6}	β_{i7}	γ_i	γ_i^*
1	0								$\frac{57}{640}$	$\frac{33}{640}$
2	$\frac{1}{18}$	$\frac{1}{18}$							0	0
3	$\frac{1}{6}$	$-\frac{1}{12}$	$\frac{1}{4}$						$-\frac{16}{65}$	$-\frac{132}{325}$
4	$\frac{2}{9}$	$-\frac{2}{81}$	$\frac{4}{27}$	$\frac{8}{81}$					$\frac{1377}{2240}$	$\frac{891}{2240}$
5	$\frac{2}{3}$	$\frac{40}{33}$	$-\frac{4}{11}$	$-\frac{56}{11}$	$\frac{54}{11}$				$\frac{121}{320}$	$-\frac{33}{320}$
6	1	$-\frac{369}{73}$	$\frac{72}{73}$	$\frac{5380}{219}$	$-\frac{12285}{584}$	$\frac{2695}{1752}$			0	$-\frac{73}{700}$
7	$\frac{8}{9}$	$-\frac{8716}{891}$	$\frac{656}{297}$	$\frac{39520}{891}$	$-\frac{416}{11}$	$\frac{52}{27}$	0		$\frac{891}{8320}$	$\frac{891}{8320}$
8	1	$\frac{3015}{256}$	$-\frac{9}{4}$	$-\frac{4219}{78}$	$\frac{5985}{128}$	$-\frac{539}{384}$	0	$\frac{693}{3328}$	$\frac{2}{35}$	$\frac{2}{35}$

and E is the difference between this approximate solution and the approximate solution having fifth order truncation precision. E is assumed to be the truncation error of y_{n+1}.

The method of automatically adjusting the step size h is as follows.

Let ε be max(required absolute precision, required relative precision $\times \left|\frac{y_n + y_{n+1}}{2}\right|$).

Obtain the minimum of $\frac{\varepsilon}{|E|}$ for each of the simultaneous differential equations, and let this value be U.

(a) If $U < 1$ (step size is too big), then update h as follows:
$h = h \max(0.85 \times \sqrt[6]{U}, 0.1)$
and obtain y_{n+1} again.

(b) If $U \geq 1$ (step size is sufficiently small), then:

i. For $1.4 \leq U < 2.4$, leave h unchanged and move on to the calculation for the next step.

ii. For $U < 1.4$ or $U \geq 2.4$, update h as follows:
$h = h \min(0.9 \times \sqrt[6]{U}, 5)$, let y_{n+1} be y_n and move on to the calculation for the next step.

Now, from $y' = f(x, y)$ that is obtained initially, let the error at the final point x_f be $y'(x_f - x)$ and use the following equations to calculate the initial value of h.

$$\begin{aligned} \varepsilon &= \max(\text{required absolute precision}, \text{required relative precision} \times y) \\ h &= \sqrt[6]{\frac{\varepsilon(x_f - x)}{y'(x_f - x)}} \end{aligned}$$

Use the minimum value of h among these for the various simultaneous differential equations.
For high-order differential equation: $y^{(d)} = f(x, y, y', \cdots, y^{(d-1)})$ let:

$$y_1 = y, y_2 = y', y_3 = y'', \cdots, y_d = y^{(d-1)}$$

and consider the following first-order simultaneous differential equations:

$$\begin{cases} y_1' &= y_2 \\ y_2' &= y_3 \\ & \vdots \\ y_d' &= f(x, y_1, \cdots, y_d) \end{cases}$$

The function of this library performs this processing automatically.
Since this method evaluates functions frequently, it is most efficient when function evaluation is not expensive and the precision requirements are not high. (See Reference Bibliography (1).)

(2) **Linear multistep method based on step size or order control quotient difference method**
Let consider differential equation $y^{(d)} = f(x, y, \cdots, y^{(d-1)})$.
Then, let $f(x_n) = f(x_n, y(x_n), \cdots, y^{(d-1)}(x_n))$ and $y_n = y(x_n)$.

(a) If $d = 1$, then the predictor p_{n+1}^0 is given by:

$$p_{n+1}^0 = y_n + \int_{x_n}^{x_{n+1}} f(x)dx$$

(b) If $d > 1$, then for $h_{n+1} = x_{n+1} - x_n$, the predictor $p_{n+1}^{(d-k)}$ is given by:

$$p_{n+1}^{(d-k)} = \sum_{i=0}^{k-1} \frac{h_{n+1}^i}{i!} y_n^{(d-k+i)} + \int_{x_n}^{x_{n+1}} \cdots \int_{x_n}^{s_2} f(s_1)ds_1 \cdots ds_k \quad (k = 1, \cdots, d) \tag{2.1}$$

where $\int ds_i$ is the integral of $y^{(d-i)}$ for the x.

Use previously calculated differential coefficients to calculate a polynomial approximation of this $f(x)$.
If we let the $(q-1)$-th order approximation passing through the q points $(x_i, f(x_i))$ $(i = n, \cdots, n-q+1)$ be $P_{q-1}(x)$, then this approximation is defined as follows by using divided differences.
If we let:

$$\begin{aligned} f[x_n] &= f(x_n) \\ f[x_n, \cdots, x_{n-i}] &= \frac{f[x_n, \cdots, x_{n-i+1}] - f[x_{n-1}, \cdots, x_{n-i}]}{x_n - x_{n-i}} \end{aligned} \tag{2.2}$$

then $P_{q-1}(x)$ is given by:

$$P_{q-1}(x) = f[x_n] + \cdots + (x - x_n) \cdots (x - x_{n-i+1}) f[x_n, \cdots, x_{n-i}] \quad (i = 0, \cdots, q-1) \tag{2.3}$$

Also, the corrector is obtained by integrating the q-th order $P_q^*(x)$ passing through the point $(x_{n+1}, f(x_{n+1}))$ in a similar manner as in expression (2.1), where $P_q^*(x)$ is given by:

$$P_q^*(x) = P_{q-1}(x) + (x - x_n)\cdots(x - x_{n-q+1})f[x_n, \cdots, x_{n-q+1}] \tag{2.4}$$

We define the following symbols for calculating expression (2.1):

$$\begin{aligned}
\tau &= \frac{x - x_n}{h_{n+1}} && (2.5)\\
\xi_i(n) &= h_n + \cdots + h_{n-i+1} = x_n - x_{n-i} && (2.6)\\
\eta_i(n) &= \frac{h_{n+1}}{\xi_i(n)} && (2.7)\\
\beta_0(n) &= 1 && (2.8)\\
\beta_i(n) &= \frac{\xi_1(n+1)\cdots\xi_i(n+1)}{\xi_1(n)\cdots\xi_i(n)} && (2.9)\\
\varphi_0(n) &= f[x_n] && (2.10)\\
\varphi_i(n) &= \xi_1(n)\cdots\xi_i(n)f[x_n, \cdots, x_{n-i}] && (2.11)
\end{aligned}$$

Then, expression (2.3) can be written as follows:

$$P_{q-1}(x) = \sum_{i=0}^{q-1}\left\{\varphi_i(n)\sum_{j=1}^{i}A_{i,j}(n)\tau^j\right\}$$

where $A_{i,j}$ is obtained by using the following recursive relation.

$$\begin{cases}
A_{i,1}(n) = \eta_i(n) & (i = 1, \cdots, q)\\
A_{i+1,j+1}(n) = \eta_{i+1}(n)\sum_{l=j}^{i}A_{l,j}(n) & (j = 1, \cdots, q-2; i = j, \cdots, q-2)
\end{cases} \tag{2.12}$$

Therefore, if we let:

$$\gamma_{ki}(n) = \begin{cases}
\frac{1}{k!} & (i = 0;\ k = 1, \cdots, d)\\
\sum_{j=1}^{i}\frac{j!}{(j+k)!}A_{ij}(n) & (i = 1, \cdots, q-1;\ k = 1, \cdots, d)
\end{cases} \tag{2.13}$$

then, expression (2.1) can be expressed as follows:

$$p_{n+1}^{(d-k)} = \sum_{i=0}^{k-1}\frac{h_{n+1}^i}{i!}y_n^{(d-k+i)} + h_{n+1}^k\sum_{i=0}^{q-1}\gamma_{k,i}(n)\varphi_i(n) \tag{2.14}$$

Similarly, from expression (2.4), the corrector $y_{n+1}^{(d-k)}$ can be expressed as follows:

$$\begin{aligned}
y_{n+1}^{(d-k)} &= p_{n+1}^{(d-k)} + h_{n+1}^k\gamma_{k,q}(n)\frac{\varphi_q(n+1)}{\beta_q(n)} \quad (k = 2, \cdots, d) && (2.15)\\
y_{n+1}^{(d-1)} &= p_{n+1}^{d-1} + h_{n+1}\sum_{i=0}^{q}\gamma_i^*(n)\varphi_i(n+1) && (2.16)
\end{aligned}$$

where $\gamma_i^*(n)$ is obtained from the following expressions.

$$\begin{cases} \gamma_0^*(n) &= 1 \\ \gamma_i^*(n) &= \dfrac{\gamma_{1,i}(n)}{\beta_i(n)} - \dfrac{\gamma_{1,i-1}(n)}{\beta_{i-1}(n)} \end{cases} \tag{2.17}$$

When $q = 1$, the predictor and corrector are given by the following expressions:

$$\begin{aligned} p_{n+1}^{(d-k)} &= \sum_{i=0}^{k-1} \frac{h_{n+1}^i}{i!} y_n^{(d-k+i)} + \frac{h_{n+1}^k}{k!} f(x_n) \\ y_{n+1}^{(d-k)} &= p_{n+1}^{(d-k)} + \frac{h_{n+1}^k}{(k+1)!}(f(x_{n+1}) - f(x_n)) \end{aligned} \tag{2.18}$$

These values are used in the first two steps of the calculation.
Order control is performed as follows.
The local discretization error is estimated according to the following equation from expression (2.16) :

$$E = |h_{n+1} \left\{ \gamma_q^*(n)\varphi_q(n+1) + \gamma_{q+1}^*(n)\varphi_{q+1}(n+1) \right\}| \tag{2.19}$$

Since the required relative precision is converted to absolute precision and the more lenient requirement between that value and the required absolute precision is taken, the required precision is obtained by using the following expression:

$$\begin{aligned} \varepsilon &= \max\left(\zeta_a, \zeta_r |y_{n+1} + h_{n+1}\frac{p'_{n+1}}{2}|\right) \\ &\quad (\zeta_a : \text{required absolute precision}, \quad \zeta_r : \text{required relative precision}) \end{aligned}$$

The convergence rate P_k of the corrector shown in expression (2.16) is defined as follows.

$$P_k = \left| \frac{\gamma_{k+1}^*(n)\varphi_{k+1}(n+1)}{\gamma_{k-1}^*(n)\varphi_{k-1}(n+1)} \right|$$

If the order q is too large, then since the propagation error becomes larger or new error components are included, P_k increases. Also, we defined various variables as follows:

$$\begin{aligned} C_{min} &= \min(p_q, p_{q-1}) \\ C_{max} &= \max(p_q, p_{q-1}) \end{aligned}$$

$$R_c = \begin{cases} 10.0 \times C_{max} & (C_{max} \leq 0.09) \\ 0.9 & (C_{min} < 0.09 < C_{max}) \\ 10.0 \times C_{min} & (0.09 \leq C_{min} \leq 0.105) \\ 1.05 & (C_{min} > 0.105) \end{cases}$$

For each simultaneous component, the order q is increased or decreased by 1 as follows:

(a) If $\frac{E}{\varepsilon} > 0.01$ and $C_{max} < 0.025$ or if $d > 1$ and $C_{max} < 0.0625$ then:
Increase q by 1.
However, for the first 9 iterations, use the following condition:
If $\frac{E}{\varepsilon} > 0.01$ and $C_{max} < 0.09$.

(b) $q > 1$, and if $C_{min} > 0.5$ or $\frac{E}{\varepsilon} < 0.001 \times P_q$ then:
Decrease q by 1.
However, for the first 9 iterations, do not decrease q.

Step size control is performed as follows.
If the number of simultaneous equations is e, then:

$$R_M = \max_j (10.0 \times \frac{E}{\varepsilon}, R_c) \quad (j = 1, \cdots, e)$$

(multiply by 10.0 to consider an extra decimal digit.)

$$\begin{aligned} \gamma &= \begin{cases} 1 + q(j_{max}) & (R_M \geq 1) \\ \max(q(j)) + \max(d(j)) & (R_M < 1) \end{cases} \\ h_{n+2} &= h_{n+1}(R_M)^{-\frac{1}{\gamma}} \end{aligned} \tag{2.20}$$

Start with an arbitrary value h for the initial step size and adjust h according to expression (2.20) until $0.125 \leq R_M \leq 3$.
The entire calculation procedure is as follows.

(a) First, use expression (2.18).

(b) Calculate $\eta_i(n)$ according to expression (2.7).

(c) Calculate $\gamma_{k,i}(n)$ of expression (2.13) according to expressions (2.5) through (2.12).

(d) Calculate $p_{n+1}^{(d-k)}$ $(k = d, d-1, \cdots, 1)$ according to expression (2.14).

(e) Calculate $\beta_i(n)$ $(i = 1, 2, \cdots, q)$ according to the following expressions:

$$\begin{aligned} \xi_i(n+1) &= h_{n+1} + \xi_{i-1}(n) \\ \beta_i(n) &= \beta_{i-1}(n) \frac{\xi_i(n+1)}{\xi_i(n)} \end{aligned}$$

Also, calculate $\gamma_i^*(n)$ according to expression (2.17). Then, approximate $\gamma_{q+1}^*(n)$ according to the following expression:

$$\gamma_{q+1}^*(n) = \frac{\{\gamma_q^*(n)\}^2}{\gamma_{q-1}^*(n)}$$

(f) Let $\varphi_0(n+1)$ be $f[x_{n+1}]$ and calculate $\varphi_{i+1}(n+1)$ according to the following expression:

$$\varphi_{i+1}(n+1) = \varphi_i(n+1) - \beta_i(n)\varphi_i(n) \quad (i = 0, \cdots, q)$$

(g) Calculate the corrector $y_{n+1}^{(d-k)}$ according to expression (2.15) or (2.16).

(h) Estimate the local discretization error according to expression (2.19) and adjust the order q.

(i) Calculate $\varphi_0(n+1)$ and modify $\varphi_i(n+1)$ $(i = 1, \cdots, q)$ according to the following expression:

$$\varphi_i(n+1) = \varphi_i(n+1) + \left\{ f(x_{n+1}, y_{n+1}, \cdots, y_{n+1}^{(d-1)}) - f(x_{n+1}, p_{n+1}, \cdots, p_{n+1}^{(d-1)}) \right\}$$

(j) Adjust the step size according to expression (2.20) and return to step (b).

This method is most efficient when function evaluations are expensive and precision requirements are severe. (See Reference Bibliography (2) and (5).)

(3) **Taylor series method (including processing corresponding to implicit equations)**
Assume that the equation to be solved is expressed as follows:

$$f(x, y, y', \cdots, y^{(d)}) = 0 \tag{2.21}$$

The following Taylor expansions can be used to obtain approximations of $y, y', \cdots, y^{(d-1)}$ at the point $x+h$:

$$\begin{aligned} y(x+h) &= y(x) + hy'(x) + \frac{h^2}{2!}y''(x) + \cdots \\ y'(x+h) &= y'(x) + hy''(x) + \frac{h^2}{2!}y'''(x) + \cdots \\ &\vdots \end{aligned}$$

To raise the precision by increasing the number of terms in the Taylor series, we create an expression that includes the higher order derivative $y^{(d+1)}$ by differentiating expression (2.21), then we differentiate again to create an expression that includes $y^{(d+2)}$, and so on. We obtain $y^{(d+1)}, y^{(d+2)}, \cdots$ from these expressions by solving a system of nonlinear simultaneous equations, and then we obtain the approximations at the point $x + h$ by assigning these values in the Taylor expansions.
The above procedure is described in more detail below.
Assume that the group of equations that are entered are as follows:

$$\begin{aligned} f_1(x, y, \cdots, y^{(d)}) &= 0 \\ f_1'(x, y, \cdots, y^{(d)}) &= f_2(x, y, \cdots, y^{(d)}, y^{(d+1)}) = 0 \\ &\vdots \end{aligned} \tag{2.22}$$

If $f_1(x, \cdots)$ or $f_2(x, \cdots)\cdots$ cannot be easily differentiated or even if they can be easily differentiated, to reduce the effort required to differentiate the entered equations, the function has a feature that automatically differentiates functions. For example, automatic difference of $f_1(x, y, \cdots, y^{(d+1)})$ is given as follows performing a central difference calculation:

$$\begin{aligned} f_2(x, y, \cdots, y^{(d)}, y^{(d+1)}) &= f_1'(x, y, \cdots, y^{(d+1)}) \\ &\simeq \frac{1}{2\delta}\{g_1(x+\delta) - g_1(x-\delta)\} \\ &\quad + \frac{y'}{2\delta}\{g_1(y+\delta) - g_1(y-\delta)\} \\ &\quad + \cdots \\ &\quad + \frac{y^{(d+2)}}{2\delta}\left\{g_1(y^{(d+1)}+\delta) - g_1(y^{(d+1)}-\delta)\right\} \\ &= 0 \end{aligned}$$

where the following notations are used.

$$\begin{aligned} g_1(x+\delta) &= f_1(x+\delta, y, \cdots, y^{(i)}, \cdots, y^{(d+1)}) \\ g_1(y^{(i)}+\delta) &= f_1(x, y, \cdots, y^{(i)}+\delta, \cdots, y^{(d+1)}) \quad (i = 0, \cdots, d+1) \end{aligned}$$

For δ, use a small value such as $\sqrt[3]{\text{Units for determining error.}}$. You can also consider higher differentials and determine $g_2(\cdots), g_3(\cdots), \cdots, g_i(\cdots)$ and create the following expression that includes $y^{(d+i)}$:

$$f_{i+1}(x, y, \cdots, y^{(d+i)}) = 0 \quad (i = 1, 2, \cdots)$$

Next, from the group of equations created in this manner, $x, y(x), y'(x), \cdots, y^{(d-1)}(x)$ are assumed to be initial values, and the following system of nonlinear simultaneous equations having $y^{(d)}(x), \cdots, y^{(d+i)}(x)$ and $y(x+h), y'(x+h), \cdots, y^{(d-1)}(x+h)$ as unknowns is created:

$$\begin{cases} y(x+h) - \left\{ y(x) + hy'(x) + \cdots + h^{d+i} \dfrac{y^{(d+i)}(x)}{(d+i)!} \right\} = 0 \\ y'(x+h) - \left\{ y'(x) + hy''(x) + \cdots + h^{d+i-1} \dfrac{y^{(d+i)}(x)}{(d+i-1)!} \right\} = 0 \\ \vdots \\ y^{(d-1)}(x+h) - \left\{ y^{(d-1)}(x) + hy^{(d)}(x) + \cdots + h^{i+1} \dfrac{y^{(d+i)}(x)}{(i+1)!} \right\} = 0 \\ f_1(x, y(x), \cdots, y^{(d)}(x)) = 0 \\ \vdots \\ f_{i+1}(x, y(x), \cdots, y^{(d+i)}(x)) = 0 \end{cases} \tag{2.23}$$

These nonlinear equations are solved and the values $y(x+h), \cdots, y^{(d-1)}(x+h)$ that were obtained are assumed to be the next initial values $y(x), \cdots, y^{(d-1)}(x)$ and $x+h$ is assumed to be the next x.
If the differential equations are simultaneous, then groups of equations are created corresponding to expression (2.23) for each equation, and the simultaneous nonlinear equations are solved by making all of these equations simultaneous. Therefore, as in the following, for example, for an implicit case that includes $y_i^{(d)}$ $(i = 1, \cdots, n)$ in multiple equation calculations:

$$\begin{aligned} &f_1^1(x, y_1, \cdots, y_1^{(d)}, \cdots, y_n, \cdots, y_n^{(d)}) = 0 \\ &\vdots \\ &f_1^n(x, y_1, \cdots, y_1^{(d)}, \cdots, y_n, \cdots, y_n^{(d)}) = 0 \end{aligned} \tag{2.24}$$

(n: Number of simultaneous equations) or even if algebraic equations $f_1^i(x, y_1, y_2, \cdots, y_n) = 0$ are taken as simultaneous equations, a solution is obtained because all of the equations that are taken to be simultaneous will be solved by using the nonlinear simultaneous equations that are processed. Also, as an extreme case, if the given differential equations include no differential terms and they are all nonlinear equations or algebraic equations, then it is assumed that an expression that includes $y(x+h)$ will not be expanded according to a Taylor expansion. The function also has a feature that obtains a solution by creating nonlinear simultaneous equations consisting only of these equations.
If the equations are simultaneous equations that include ordinary differential equations, then integration will continue with a step size of h up to the final point x. Finally, to obtain the solution of $y_i^{(d)}(x+h)$, the value of $y_i^{(d-i)}(x+h)$ that was obtained is entered in the following nonlinear equations:

$$\begin{aligned} &f_1^1(x+h, y_1(x+h), \cdots, \underline{y_1^{(d)}(x+h),} \cdots, \underline{y_n^{(d)}(x+h)}) = 0 \\ &\vdots \\ &f_1^n(x+h, y_1(x+h), \cdots, \underline{y_1^{(d)}(x+h),} \cdots, \underline{y_n^{(d)}(x+h)}) = 0 \end{aligned} \tag{2.25}$$

where, terms underlined by ___ are unknowns. The nonlinear simultaneous equations are solved as follows. If given equation is only one equation, which is not simultaneous, then consider the following iteration method for obtaining the value of x where $f(x) = 0$.

$$x_{n+1} = x_n + 2^{\frac{p-3r-1}{3}} S \sinh^{-1}\{f(x_n)\} \quad (S = \pm 1) \tag{2.26}$$

S is assumed to be -1 or 1 as follows.
When $f(x_n) < 0$ (if $f(x)$ increases monotonously, then: $x > x_n$), then $S = -1$.
When $f(x_n) > 0$ (if $f(x)$ decreases monotonously, then: $x < x_n$), then $S = 1$.
The initial values of r and p are assumed to be zero. r performs deceleration control by increasing by 1 each time the sign of the calculated solution of $f(x_n)$ changes, and p performs acceleration control by increasing by 1 when the sign of the calculated solution does not change.
When the following simultaneous equations are solved,

$$\begin{cases} f_1(x_1, x_2, \cdots, x_n) = 0 \\ f_n(x_1, x_2, \cdots, x_n) = 0 \end{cases}$$

then, first, x_1 is assumed to be the unknown in the f_1 expression, and expression correspond to expression (2.26) for f_1 is iterated once with x_2 through x_n taken as initial values. Next, x_2 is assumed to be the unknown in the f_2 expression, and expression correspond to expression (2.26) for f_2 is iterated once with x_1 taken to be the value of the previous calculation and x_3 through x_n taken to be initial values. When the calculations are performed in this way up to f_n, control returns again to f_1 and the procedure described above is repeated. In this way, the calculated values will converge to the correct solution. The initial value $S_i \quad (i = 1, \cdots, n)$ has been determined to be $S_i = -1$ if the slope of f_i relative to x_i is positive and $S_i = 1$ if the slope is negative. However, even if the initial value has been determined in this way, the slope may reverse direction locally and the direction of investigation may be incorrect. Therefore, if $|f_i| > 10,000$ after iterating nine or more times, the direction of investigation is considered to be incorrect, $S_i = -S_i$ is assumed, the initial values are reset, and the simultaneous equations are solved again. Also, if the direction of investigation is considered to be incorrect again even when the direction of investigation was modified during the previous calculations, then the calculation is considered to be impossible, and processing is aborted.
The function of this library assumes that convergence has occurred when

$$x_{n+1} < |\varepsilon x_n| + \varepsilon \text{ and } |f_i(\cdots)| < |10.0 \times \varepsilon x_n| + \varepsilon$$

where the maximum iteration count of these nonlinear simultaneous equations is assumed to be 100 and the convergence decision value ε is given by:

$$\varepsilon = \left\{ \begin{array}{l} \text{Single precision:} 10^{-5} \\ \text{Double precision:} 10^{-12} \end{array} \right\}.$$

Since this method must solve many simultaneous nonlinear equations each time h advances, efficiency is poor. However, it is effective for ordinary differential equations that are simultaneous with algebraic or nonlinear equations or an implicit problem since they cannot be solved by other methods. Also, even when all the equations are nonlinear simultaneous equations, a solution is obtained rather efficiently by only entering the equations to be solved without having to create a function for calculating partial derivatives as required when using Newton's method.
(See Reference Bibliography (3).)

(4) **Gear's method for stiff problems**
This function uses Gear's 1st order through 5th order method. Selection of the order and step size are automatically controlled.

(a) Predictor and corrector calculations

We assume that the differential equation is given in the form shown below, where $\boldsymbol{y}$ is assumed to be an N-dimensional vector:

$$\boldsymbol{y}' = \boldsymbol{f}(x, \boldsymbol{y})$$

Now, we assume that calculated solutions $\boldsymbol{y}_i \quad (i = 0, \cdots, n-1)$ up to $x = x_{n-1}$ have been obtained and that we are to obtain the calculated solution at $x = x_n$. At this time, a q-th order interpolation polynomial vector $\boldsymbol{p}_{n-1}(x)$ that satisfies the following conditions can be created:

$$\begin{aligned} \boldsymbol{p}_{n-1}(x_{n-i}) &= \boldsymbol{y}_{n-i} \quad (i = 1, \cdots, q) \\ \boldsymbol{p}'_{n-1}(x_{n-1}) &= \boldsymbol{f}(x_{n-1}, \boldsymbol{y}_{n-1}) \end{aligned}$$

The basic concept of this method is to try to determine $\boldsymbol{y}_n$ so that when the q-th order interpolation polynomial vector $\boldsymbol{p}_n(x)$ is created by using the calculated solution $\boldsymbol{y}_n$ at $x = x_n$, it satisfies the following conditions:

$$\begin{aligned} \boldsymbol{p}_n(x_{n-i}) &= \boldsymbol{y}_{n-i} \quad (i = 0, \cdots, q) \\ \boldsymbol{p}'_n(x_n) &= \boldsymbol{f}(x_n, \boldsymbol{y}_n) \end{aligned}$$

Information related to $\boldsymbol{p}_{n-1}(x)$ is retained in the form of the following matrix, which was conceived by **Nordsieck**:

$$Z_{n-1} = \left[\boldsymbol{y}_{n-1}, h\boldsymbol{y}'_{n-1}, h^2 \frac{\boldsymbol{y}''_{n-1}}{2!}, \cdots, h^q \frac{\boldsymbol{y}^{(q)}_{n-1}}{q!} \right]$$

where:

$$\begin{aligned} \boldsymbol{y}^{(q)}_{n-1} &= \boldsymbol{p}^{(q)}_{n-1}(x_{n-1}) \\ h &= x_n - x_{n-1} \end{aligned}$$

The predictor $Z_{n(0)}$ of Z_n is defined as follows:

$$\begin{aligned} Z_{n(0)} &= \left[\boldsymbol{y}_{n(0)}, h\boldsymbol{y}'_{n(0)}, h^2 \frac{\boldsymbol{y}''_{n(0)}}{2!}, \cdots, h^q \frac{\boldsymbol{y}^{(q)}_{n(0)}}{q!} \right] \\ \boldsymbol{y}^{(q)}_{n(0)} &= \boldsymbol{p}^{(q)}_{n-1}(x_n) \end{aligned}$$

$Z_{n(0)}$ can be calculated by using the following expression:

$$Z_{n(0)} = Z_{n-1} A \tag{2.27}$$

where, A is a Pascal triangle matrix whose i, j-th component $a_{i,j}$ is defined as follows:

$$a_{i,j} = \begin{cases} 0 & (i < j) \\ \dfrac{i!}{j!(i-j)!} & (i \geq j) \end{cases} \qquad (i, j = 0, \cdots, q)$$

Now, the polynomial $L_n(s)$ for s is defined as follows:

$$\begin{aligned} L_n(s) &= \prod_{i=1}^{q} (1 + \frac{s}{d_i}) \\ d_i &= \frac{x_n - x_{n-i}}{h} \end{aligned}$$

and the coefficient vector $\boldsymbol{l}$ of $L_n(s)$ is defined as follows:

$$\begin{aligned} \boldsymbol{l} &= [l_0, l_1, \cdots, l_q] \\ L_n(s) &= \sum_{i=0}^{q} l_i s^i \end{aligned}$$

At this time, the following relationship can be shown to occur:

$$\begin{aligned} Z_n &= Z_{n(0)} + \boldsymbol{e}_n \boldsymbol{l} \\ \boldsymbol{e}_n &= \boldsymbol{y}_n - \boldsymbol{y}_{n(0)} \end{aligned} \tag{2.28}$$

Since the following relationship is obtained by writing the first column of this:

$$h\boldsymbol{y}'_n = h\boldsymbol{y}'_{n(0)} + (\boldsymbol{y}_n - \boldsymbol{y}_{n(0)})l_1$$

$\boldsymbol{y}_n$ is calculated as the root of $\boldsymbol{g}(\boldsymbol{y}) = 0$ if $\boldsymbol{g}(\boldsymbol{y})$ is defined as follows:

$$\boldsymbol{g}(\boldsymbol{y}) = \boldsymbol{y} - \boldsymbol{y}_{n(0)} - (\frac{h}{l_1})(\boldsymbol{f}(x_n, \boldsymbol{y}) - \boldsymbol{y}_{n(0)'})$$

To solve the equation $\boldsymbol{g}(\boldsymbol{y}) = 0$, Newton's method is used with $\boldsymbol{y} = \boldsymbol{y}_{n(0)}$ as the starting value. That is, the calculation is performed using the following recursive relation:

$$\begin{aligned} \boldsymbol{y}_{n(m+1)} &= \boldsymbol{y}_{n(m)} - P_m^{-1} \boldsymbol{g}(\boldsymbol{y}_{n(m)}) \\ P_m &= 1 - \frac{h}{l_1} J(x_n, \boldsymbol{y}_{n(m)}) \end{aligned}$$

where, $J(x, \boldsymbol{y})$, which is the Jacobian matrix of $\boldsymbol{f}(x, \boldsymbol{y})$, is defined as follows:

$$J(x, \boldsymbol{y}) = \begin{bmatrix} \dfrac{\partial f_1(x, \boldsymbol{y})}{\partial y_1} & \cdots & \dfrac{\partial f_1(x, \boldsymbol{y})}{\partial y_N} \\ \vdots & & \\ \dfrac{\partial f_N(x, \boldsymbol{y})}{\partial y_1} & \cdots & \dfrac{\partial f_N(x, \boldsymbol{y})}{\partial y_N} \end{bmatrix}$$

To save calculation time, the program directly uses the Jacobian matrix calculated for the previous iteration as much as possible. Also, corrector iterations are performed at most three times.

(b) Order and step size determination

This section describes how error is evaluated and how step size and order are controlled.

If we let the local discretization absolute error be $E_n(q)$ and the local discretization relative error be $R_n(q)$, then the function decides that the calculated solution is to be accepted as follows by using the two user-assigned parameters, "Required local relative precision" E_a and "Required local absolute precision" E_r :

$$\|E_n(q)\| \leq E_a \quad or \quad \|R_n(q)\| \leq E_r \tag{2.29}$$

where, $\|\ \|$ indicates the Max norm. Also, the relative error is the ratio of the absolute error to the maximum value of the calculated solution up to the current time.

If the calculated solution satisfied the condition shown in (2.29), then the step size and order to be used in the next step are selected as follows. In addition to the error values described above, the errors $E_n(q-1)$ and $R_n(q-1)$ at order $q-1$ and the errors $E_n(q+1)$ and $R_n(q+1)$ at order $q+1$ are calculated. These are used to calculate the maximum step size that is permitted. That is, the maximum value among the $\eta_i \quad (i = 1, \cdots, 6)$ shown below is assumed to be the rate of increase of h,

and the order used for the error calculation at that time is assumed to be the order for the next step.

$$\eta_1 = \frac{\sqrt[q]{\frac{E_a}{\|E_n(q-1)\|}}}{1.3}$$

$$\eta_2 = \frac{\sqrt[q]{\frac{E_r}{\|R_n(q-1)\|}}}{1.3}$$

$$\eta_3 = \frac{\sqrt[(q+1)]{\frac{E_a}{\|E_n(q)\|}}}{1.2}$$

$$\eta_4 = \frac{\sqrt[(q+1)]{\frac{E_r}{\|R_n(q)\|}}}{1.2}$$

$$\eta_5 = \frac{\sqrt[(q+2)]{\frac{E_a}{\|E_n(q+1)\|}}}{1.4}$$

$$\eta_6 = \frac{\sqrt[(q+2)]{\frac{E_r}{\|R_n(q+1)\|}}}{1.4}$$

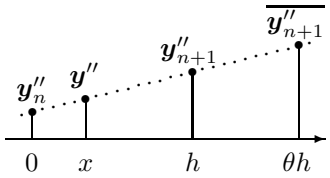
However, if the calculated solution did not satisfy the condition shown in (2.29), then the step size is decreased and the current step is performed again.

The integration starting process, which is self starting, is calculated according to a linear formula.

(5) **Wilson's θ method**

We assume that $\boldsymbol{y}''$ varies rectilinearly at $x = 0$ and $x = h$ as follows:

$$\boldsymbol{y}'' = \boldsymbol{y}''_n \frac{h-x}{h} + \boldsymbol{y}''_{n+1}$$



We integrate this expression with respect to x to obtain the following:

$$\boldsymbol{y}' = \boldsymbol{y}'_n + \frac{\boldsymbol{y}'_n}{h}\left(hx - \frac{x^2}{2}\right) + \frac{\boldsymbol{y}''_{n+1}}{h}\frac{x^2}{2}$$

$$\boldsymbol{y} = \boldsymbol{y}_n + \boldsymbol{y}'_n x + \frac{\boldsymbol{y}''_n}{h}\left(\frac{hx^2}{2} - \frac{x^3}{6}\right) + \frac{\boldsymbol{y}''_{n+1}}{h}\frac{x^3}{6}$$

If we let $x = h$ in the above expressions, then $\boldsymbol{y}'$ and $\boldsymbol{y}$ become $\boldsymbol{y}'_{n+1}$ and $\boldsymbol{y}_{n+1}$. That is, we obtain the following expressions:

$$\boldsymbol{y}'_{n+1} = \boldsymbol{y}'_n + \boldsymbol{y}''_n \frac{h}{2} + \boldsymbol{y}''_{n+1}\frac{h}{2} \tag{2.30}$$

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \boldsymbol{y}'_n h + \boldsymbol{y}''_n \frac{h^2}{3} + \boldsymbol{y}''_{n+1}\frac{h^2}{6} \tag{2.31}$$

If we substitute (2.30) and (2.31) into the original equation, then we obtain:

$$M\boldsymbol{y}''_{n+1} + C\left\{\boldsymbol{y}'_n + (\boldsymbol{y}''_n + \boldsymbol{y}''_{n+1})\frac{h}{2}\right\} + K\left\{\boldsymbol{y}_n + \boldsymbol{y}'h + (2\boldsymbol{y}''_n + \boldsymbol{y}''_{n+1})\frac{h^2}{6}\right\} = \boldsymbol{p}(h)$$

If we solve this for $\boldsymbol{y}''_{n+1}$, we obtain the following simultaneous linear equations:

$$\begin{aligned}&\left\{M+\frac{h}{2}C+\frac{h^2}{6}K\right\}\boldsymbol{y}''_{n+1}\\&=\left[\boldsymbol{p}(h)-C\left\{\boldsymbol{y}'_n+\boldsymbol{y}''_n\frac{h}{2}\right\}-K\left\{\boldsymbol{y}_n+\boldsymbol{y}'_n h+\boldsymbol{y}''_n\frac{h^2}{3}\right\}\right]\end{aligned}\tag{2.32}$$

However, since the solution often is unstable if we solve this directly, we solve for $\overline{\boldsymbol{y}''_{n+1}}$ at the point θh, which is obtained by multiplying the step size h by θ, and then obtain $\boldsymbol{y}''_{n+1}$ according to the following expression:

$$\boldsymbol{y}''_{n+1}=\boldsymbol{y}''_n+\frac{\overline{\boldsymbol{y}''_{n+1}}-\boldsymbol{y}''_n}{\theta}\tag{2.33}$$

It is known that θ should be at least 1.37. However, if this value is too large, the truncation error will increase and precision will worsen. This increase in error appears rather dramatically, for example, even for $\theta=2$. Wilson recommended that 1.4 be used as a practical value for θ.
The discussion above is summarized in the following procedure.

(a) Obtain the first initial value $\boldsymbol{y}''_n$ by solving the following simultaneous linear equations:

$$M\boldsymbol{y}''_n=\{\boldsymbol{p}(x)-C\boldsymbol{y}'_n-K\boldsymbol{y}\}\tag{2.34}$$

However, if the simultaneous linear equations cannot be solved because a zero is included in the diagonal components of the matrix M, then let $\boldsymbol{y}''_n$ be zero, divide the step size h by 8, use Wilson's θ method shown in steps (b) through (d) to obtain $\boldsymbol{y}''_{n+1}$, $\boldsymbol{y}'_{n+1}$ and $\boldsymbol{y}_{n+1}$ at the point that is offset ahead by h, and jump to step (d).

(b) Obtain $\overline{\boldsymbol{y}''_{n+1}}$ at the point that is offset ahead by θh by solving the following simultaneous linear equations:

$$\begin{aligned}\left\{M+\frac{\theta h}{2}C+\frac{(\theta h)^2}{6}K\right\}\overline{\boldsymbol{y}''_{n+1}}\quad&=\boldsymbol{p}(x)+(\boldsymbol{p}(x+h)-\boldsymbol{p}(x))\theta-C\left\{\boldsymbol{y}'_n+\boldsymbol{y}''_n\frac{\theta h}{2}\right\}\\&\quad-K\left\{\boldsymbol{y}_n+\boldsymbol{y}'_n\theta h+\boldsymbol{y}''_n\frac{(\theta h)^2}{3}\right\}\end{aligned}\tag{2.35}$$

Now, $\left\{M+\frac{\theta h}{2}C+\frac{(\theta h)^2}{6}K\right\}$ is fixed as long as the step size h does not change. Therefore, first, an LU decomposition is performed once and then, if forward and back substitution are performed while recreating the right-hand side term for each x, $\overline{\boldsymbol{y}''_{n+1}}$ is obtained at each x.
Since the initial value of $\boldsymbol{y}''_n$ is considered to be inappropriately set if the difference between $\overline{\boldsymbol{y}''_{n+1}}$ and $\boldsymbol{y}''_n$ is considerably large such as on the order of 1/(Unit for determining error) even at a single component, then let $\boldsymbol{y}''_n$ be zero, divide the step size h by 8, use Wilson's θ method shown in steps (b) through (d) to obtain $\overline{\boldsymbol{y}''_{n+1}}$, $\boldsymbol{y}'_n$ and $\boldsymbol{y}_n$ at the point that is offset ahead by h, and jump to step (d).

(c) Obtain $\boldsymbol{y}''_{n+1}$ according to expression (2.33).

(d) Obtain $\boldsymbol{y}_{n+1}$ according to expression (2.31).

(e) Obtain $\boldsymbol{y}'_{n+1}$ according to expression (2.30).

(f) Let $\boldsymbol{y}''_n=\boldsymbol{y}''_{n+1}$, $\boldsymbol{y}'_n=\boldsymbol{y}'_{n+1}$ and $\boldsymbol{y}_n=\boldsymbol{y}_{n+1}$ and execute the procedure again starting from (b) for the next step.

This method can be used, for example, for equations of motion with M corresponding to the mass matrix, C to the damping matrix, K to the stiffness matrix, $\boldsymbol{p}(x)$ to the external force at time x, x to time, $\boldsymbol{y}''$ to acceleration, $\boldsymbol{y}'$ to velocity, and $\boldsymbol{y}$ to position. For an earthquake response analysis, $\boldsymbol{p}(x)$ can be considered to be $\boldsymbol{p}(x)=-M\boldsymbol{y}''_e$, where $\boldsymbol{y}''_e$ is the acceleration of the ground.

2.1.2.2 Ordinary Differential Equations (Boundary Value Problems)

(1) **Multipoint shooting method**

This method solves the boundary value problem by selecting two or more shooting points within the interval and searching for the initial value at the previous shooting point that makes the residual at that point zero.
Let the differential equations be expressed as:

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \cdots, y_n) \\ y_2' &= f_2(x, y_1, y_2, \cdots, y_n) \\ &\vdots \\ y_n' &= f_n(x, y_1, y_2, \cdots, y_n) \end{aligned} \tag{2.36}$$

and the boundary conditions as:

$$\begin{aligned} g_1(y_1(a), y_2(a), \cdots, y_n(a), y_1(b), y_2(b), \cdots, y_n(b)) &= 0 \\ g_2(y_1(a), y_2(a), \cdots, y_n(a), y_1(b), y_2(b), \cdots, y_n(b)) &= 0 \\ &\vdots \\ g_n(y_1(a), y_2(a), \cdots, y_n(a), y_1(b), y_2(b), \cdots, y_n(b)) &= 0 \end{aligned} \tag{2.37}$$

(Where, $x = a$:left-hand side boundary; $x = b$:right-hand side boundary)

Let the number of shooting points be n_x, and let the vector that approximates the strict solutions $y_j(x_i)$ $(j = 1, 2, \cdots, n)$ at shooting point x_i $(i = 1, 2, \cdots, n_x - 1)$ be expressed as:

$$\boldsymbol{u}_i = (u_{(i)1}, u_{(i)2}, \cdots, u_{(i)n})$$

Obtain the following equation from (2.36) by letting $\hat{y}_j(x_i) = u_{(i)j}$.

$$\hat{y}_j' = f_j(x, \hat{y}_1, \hat{y}_2, \cdots, \hat{y}_n) \tag{2.38}$$

From (2.38), let the vector obtained by integrating up to x_{i+1} with the vector $\boldsymbol{u}_i$ as the initial value be $\hat{y}_j(x_{i+1})$. Use the Runge-Kutta-Verner method for the initial value problem calculation at this time. (See Section 2.1.2)
The residuals are expressed as follows:

$$\begin{aligned} r_{(i)j} &= \hat{y}_j(x_{i+1}) - u_{(i+1)j} \\ r_{(n_x)j} &= g_j(u_{(1)1}, u_{(1)2}, \cdots, u_{(1)n}, u_{(n_x)1}, u_{(n_x)2}, \cdots, u_{(n_x)n}) \end{aligned} \tag{2.39}$$

Finding the vector $\boldsymbol{u}_i$ that sets the residuals of (2.39) to zero is basic here.
Use Newton's method to solve these $n \times n_x$ nonlinear simultaneous equations.
Let $\Delta\boldsymbol{u}_i$ be the modified vector of vector $\boldsymbol{u}_i$. This modified vector $\Delta\boldsymbol{u}_i$ is obtained by solving the first order simultaneous equations:

$$\begin{bmatrix} A_1 & -I & 0 & & 0 & 0 \\ 0 & A_2 & -I & & 0 & 0 \\ 0 & 0 & A_3 & \cdots & 0 & 0 \\ & & \vdots & & & \\ 0 & 0 & 0 & & A_{n_x-1} & -I \\ G_1 & 0 & 0 & & 0 & G_{n_x} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{u}_1 \\ \Delta\boldsymbol{u}_2 \\ \Delta\boldsymbol{u}_3 \\ \vdots \\ \Delta\boldsymbol{u}_{n_x-1} \\ \Delta\boldsymbol{u}_{n_x} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{r}_1 \\ -\boldsymbol{r}_2 \\ -\boldsymbol{r}_3 \\ \vdots \\ -\boldsymbol{r}_{n_x-1} \\ -\boldsymbol{r}_{n_x} \end{bmatrix} \tag{2.40}$$

Where, elements of matrix A_i, G_1 and G_{n_x} is given as follows:

$$\begin{aligned}(A_i)_{jk} &= \frac{\partial \hat{y}_j(x_{i+1})}{\partial u_{(i)k}} \\ (G_1)_{jk} &= \frac{\partial g_j}{\partial u_{(1)k}} \\ (G_{n_x})_{jk} &= \frac{\partial g_j}{\partial u_{(n_x)k}}\end{aligned} \tag{2.41}$$

Since it is difficult to directly solve the first order simultaneous equations shown in (2.40), the calculation is performed as follows:

$A_{n_x} \leftarrow G_1$
$\Delta \boldsymbol{u}_{n_x} \leftarrow -\boldsymbol{r}_{n_x}$
for $i = 1, \cdots, n_x - 1$
  $A_{n_x} \leftarrow A_{n_x} A_i^{-1}$
  $\Delta \boldsymbol{u}_{n_x} \leftarrow \Delta \boldsymbol{u}_{n_x} + A_{n_x} \boldsymbol{r}_i$

$A_{n_x} \leftarrow A_{n_x} + G_{n_x}$
$\Delta \boldsymbol{u}_{n_x} \leftarrow A_{n_x}^{-1} \Delta \boldsymbol{u}_{n_x}$
for $i = n_x - 1, \cdots, 1$
  $\Delta \boldsymbol{u}_i \leftarrow A_i^{-1}(\Delta \boldsymbol{u}_{i+1} - \boldsymbol{r}_i)$

where, A_i^{-1} $(i = 1, 2, \cdots, n_x - 1)$ is obtained as follows, without directly calculating the inverse matrix. If equation (2.38) is differentiated with respect to $u_{(i)k}$, the following is obtained:

$$\frac{d}{dx}\left(\frac{\partial \hat{y}_j}{\partial u_{(i)k}}\right) = \sum_{p=1}^{n} \frac{\partial f_j}{\partial \hat{y}_p} \frac{\partial \hat{y}_p}{\partial u_{(i)k}} \tag{2.42}$$

In this equation, let:

$$(\hat{A}_i(x))_{jk} = \frac{\partial \hat{y}_j(x)}{\partial u_{(i)k}} \quad (\hat{A}_i(x_i) = I, A_i(x_{i+1}) = A_i)$$

to express equation (2.42) as follows:

$$\frac{d}{dx}\hat{A}_i = J\hat{A}_i \quad \left((J)_{jp} = \frac{\partial f_j(x, y_1, y_2, \cdots, y_n)}{\partial y_p}\right)$$

The following equation can be derived from this:

$$\frac{d}{dx}\hat{A}_i^{-1} = -J\hat{A}_i^{-1} \quad \left(\hat{A}_i^{-1}(x_i) = I\right) \tag{2.43}$$

Obtain $A_i^{-1} = \hat{A}_i^{-1}(x_{i+1})$ by integrating equation (2.43) up to x_{i+1}. That is:

$$\begin{aligned}A_i^{-1} &= \exp((x_i - x_{i+1})J) \\ &\simeq I + B + \frac{B^2}{2!} + \frac{B^3}{3!} + \frac{B^4}{4!} + \frac{B^5}{5!}\end{aligned}$$

(where, $B = (x_i - x_{i+1})J$)
From the above, the procedure for calculating vector $\boldsymbol{u}_i$ $(i = 1, 2, \cdots, n_x)$ is as follows.

(a) As the initial value of $\boldsymbol{u}_i$, set all components to 0.1.

(b) Determine the shooting points.
Determine the smallest integer i for which $i \geq b-a$. The number of shooting points n_x is given by the following formula:

$$n_x = \min(2 \times i + 4, 50)$$

Assign the shooting points equally spaced within the interval according to this value of n_x.

(c) Calculate A_i^{-1} $(i = 1, 2, \cdots, n_x - 1)$.
For the condition number $\|A_i\|\|A_i^{-1}\|$ of matrix A_i, if the relationship:

$$\|A_i\|\|A_i^{-1}\| \geq 2.5$$

is satisfied, add the midpoint of x_i and x_{i+1} as a new shooting point to the previously used shooting points, and then calculate A_i^{-1} again.

(d) Calculate G_1 and G_{n_x}.

(e) Calculate residual $\boldsymbol{r}_i$ $(i = 1, 2, \cdots, n_x)$.

(f) Calculate modified vector $\Delta\boldsymbol{u}_i$ $(i = 1, 2, \cdots, n_x)$.

(g) Update vector $\boldsymbol{u}_i$ $(i = 1, 2, \cdots, n_x)$ by using the following equation:

$$\boldsymbol{u}_i \leftarrow \boldsymbol{u}_i + \Delta\boldsymbol{u}_i$$

(h) Determine convergence
Let ε_r be the required relative precision and ε_a be the required absolute precision. (For single precision, let ε_r and ε_a be 5 times the required local precisions used to solve the initial value problem, and for double precision, let them be 10 times the required local precisions.) Consider that the sequence has converged if:

$$\max(|r_{(i)j}|, |\Delta u_{(i)j}|) < \max(\varepsilon_a, \varepsilon_r |u_{(i)j}|)$$

are satisfied for all i $(i = 1, 2, \cdots, n_x)$ and j $(j = 1, 2, \cdots, n)$. If these conditions are not satisfied, return to step (e) and perform the calculations again.
For a nonlinear problem, perform the following calculations for convergence stability.
Multiply the nonlinear calculation part of the given problem by α. At first, linearize the problem by setting the parameter α to zero. When the number of iterations is 10 or the solution has converged, return the parameter α to 1, and solve the original nonlinear problem using the solution at that time as the initial value. When solving the nonlinear problem, recalculate A_i^{-1} for each iteration.

(2) **Collocation method**

This method selects a point x within an interval and makes the residual at that point be zero. At this time, it uses a spline function based on a B-spline to represent the approximate solution of the ordinary differential equations.
Let the differential equation to be solved be expressed as:

$$a_1(x)y^{(m)} + a_2(x)y^{(m-1)} + \cdots + a_{m+1}(x)y + a_{m+2}(x) = 0 \tag{2.44}$$

the boundary conditions at the left-hand side boundary $x = a$ be expressed as:

$$y^{(d_i)}(a) = c_1, y^{(d_2)}(a) = c_2, \cdots, y^{(d_h)} = c_h \tag{2.45}$$

and the boundary conditions at the right-hand side boundary $x = b$ be expressed as:

$$y^{(d_{h+1})}(b) = c_{h+1}, y^{(d_{h+2})}(b) = c_{h+2}, \cdots, y^{(d_m)} = c_m \tag{2.46}$$

where, d_1 through d_m are related as follows:

$$0 \leq d_1 < d_2 < \cdots < d_h < m$$
$$0 \leq d_{h+1} < d_{h+2} < \cdots < d_m < m$$

Now, take n_x selected points x_i $(i = 1, 2, \cdots, n_x)$ within the interval for which the solution is to be obtained, obtain the B-spline function $u(x)$ that satisfies $y(x_i) = u(x_i)$, and consider this to be the approximate solution. If the $(k-1)$st order B-spline basis is represented by $B_{i,k}(x)$, then $u(x)$ can be represented as follows:

$$u(x) = \sum_{i=1}^{n_x} e_i B_{i,k}(x) \tag{2.47}$$

These e_i are unknown coefficients that must be obtained. Use the following equations to obtain them. From the boundary conditions at the left-hand side boundary $x = a$ shown in (2.45),

$$\sum_{i=1}^{n_x} e_i B_{i,k}^{(d_p)}(a) = c_p \quad (p = 1, 2, \cdots, h) \tag{2.48}$$

Next, substitute equation (2.47) into equation (2.44) to obtain:

$$\sum_{l=1}^{m+1} \sum_{i=1}^{n_x} e_i (a_l(x) B_{i,k}^{(m+1-l)}(x)) + a_{m+2}(x) = 0 \tag{2.49}$$

From the boundary conditions at the right-hand side boundary $x = b$ shown in (2.46),

$$\sum_{i=1}^{n_x} e_i B_{i,k}^{(d_p)}(b) = c_q \quad (q = h+1, \cdots, m) \tag{2.50}$$

The above equations can be expressed as simultaneous linear equations:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{2.51}$$

where, the matrix A, the right-hand side vector $\boldsymbol{b}$ and the vector of unknowns $\boldsymbol{x}$ are given as follows:

$$A = \begin{bmatrix} B_{1,k}^{(d_1)}(a) & \cdots & B_{n_x,k}^{(d_1)}(a) \\ \vdots & & \vdots \\ B_{1,k}^{(d_h)}(a) & \cdots & B_{n_x,k}^{(d_h)}(a) \\ \sum_{l=1}^{m+1} a_l(x_{h+1}) B_{1,k}^{(m+1-l)}(x_{h+1}) & \cdots & \sum_{l=1}^{m+1} a_l(x_{h+1}) B_{n_x,k}^{(m+1-l)}(x_{h+1}) \\ \vdots & & \vdots \\ \sum_{l=1}^{m+1} a_l(x_{n_x-m+h}) B_{1,k}^{(m+1-l)}(x_{n_x-m+h}) & \cdots & \sum_{l=1}^{m+1} a_l(x_{n_x-m+h}) B_{n_x,k}^{(m+1-l)}(x_{n_x-m+h}) \\ B_{1,k}^{(d_{h+1})}(b) & \cdots & B_{n_x,k}^{(d_{h+1})}(b) \\ \vdots & & \vdots \\ B_{1,k}^{(d_m)}(b) & \cdots & B_{n_x,k}^{(d_m)}(b) \end{bmatrix}$$

$$\boldsymbol{x} = \begin{bmatrix} e_1 \\ \vdots \\ e_h \\ e_{h+1} \\ \vdots \\ e_{n_x-m+h} \\ e_{n_x-m+h+1} \\ \vdots \\ e_{n_x} \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} c_1 \\ \vdots \\ c_h \\ -a_{m+2}(x_{h+1}) \\ \vdots \\ -a_{m+2}(x_{n_x-m+h}) \\ c_{h+1} \\ \vdots \\ c_m \end{bmatrix}$$

From the above, the procedure for calculating the approximate solution $u(x)$ is as follows.

(a) Set the selected points
Determine the smallest integer i for which $i \geq b - a$. The number of selected points n_x is given by the following formula:

$$n_x = \max(\min(5 \times i + 1, 101), m + 2)$$

Assign the selected points equally spaced within the interval according to this value of n_x.

(b) Set the degree of the spline
Set the degree of the spline proportionate to the number of selected points. Initially, determine the smallest integer i for which $i \geq \dfrac{n_x}{10}$. At this time, let the order of the spline j_s be given by:

$$j_s = \max(i + 3, m)$$

(c) Set nodes
Let $k = j_s + 1$. Then the required number of nodes is $n_x + k$. Set these nodes as follows:

$$\begin{aligned} &q_1 = q_2 = \cdots = q_k = x_1 \\ &q_{i+k} = \frac{x_i + x_{i+k}}{2} \quad (i = 1, 2, \cdots, n_x - k) \\ &q_{n_x+1} = q_{n_x+2} = \cdots = q_{n_x+k} = x_{n_x} \end{aligned}$$

(d) Set the B-spline at the selected points
The $(k-1)$st order B-spline $B_{j,k}(x_i) \quad (j = 1, 2, \cdots, n_x)$ at selected points $x_i \quad (i = 1, 2, \cdots, n_x)$ is represented by the recursive relation:

$$B_{j,k}(x) = (\frac{x_i - q_i}{q_{j+k-1} - q_j})B_{j,k-1}(x_i) + (\frac{q_{j+k} - x_i}{q_{j+k} - q_{j+1}})B_{j+1,k-1}(x_i) \tag{2.52}$$

Set the following 0th order B-spline as the starting value:

$$B_{j,1}(x_i) = \begin{cases} 1 & (q_j \leq x_i < q_{j+1}) \\ 0 & (x_i < q_j, x_i \geq q_{j+1}) \end{cases}$$

(e) Differential the B-spline at the selected points
If (2.52) is differentiated in terms of selected point $x_i \quad (i = 1, 2, \cdots, n_x)$, the following recursive relation is obtained:

$$B'_{j,k}(x_i) = (k-1)(\frac{B_{j,k-1}(x_i)}{q_{j+k-1} - q_j} - \frac{B_{j+1,k-1}(x_i)}{q_{j+k} - q_{j+1}}) \tag{2.53}$$

Sequentially differentiate both sides in a similar manner and obtain the higher order differentials of the B-spline by sequentially eliminating the first order differentials according to (2.53).

(f) Obtain the coefficient matrix of the simultaneous equations shown in (2.51) and determine e_i
Substitute the values:

$$B_{j,k}(x_i), B'_{j,k}(x_i), \cdots, B^{(m)}_{j,k}(x_i) \quad (i = 1, 2, \cdots, n_x; j = 1, 2, \cdots, n_x)$$

that were calculated in steps (d) and (e) into the coefficient matrix on the left side of the simultaneous equations shown in (2.51).
The linear combination coefficients $e_i \quad (i = 1, 2, \cdots, n_x)$ are obtained by creating the simultaneous equations shown in (2.51) according to the above and solving them.

(g) Calculate the approximate solution
From (2.47), obtain the approximate solution that passes through the strict solution at the n_x selected points.

(h) Determine convergence
Determine an index $\gamma_j \quad (j = 1, 2, \cdots, n_x - 1)$ that can take the proportion that the approximate solution curve is separated from the strict solution curve

$$\gamma_j = \left| \frac{\Delta\delta_j}{\delta Y_j} \right|$$

where:
$|\delta Y_j|$ is Maximum absolute value among the terms of the ordinary differential equation calculated at the midpoint of selected points x_j and x_{j+1}

$$|\delta Y_j| \quad = \quad \max_{l=1,2,\cdots,m+1} \left(|\sum_{i=1}^{n_x} e_i a_l(\frac{x_j + x_{j+1}}{2}) B^{(m+1-l)}_{i,k}(\frac{x_j + x_{j+1}}{2})|, |a_{m+2}(\frac{x_j + x_{j+1}}{2})| \right),$$

$|\Delta\delta_j|$ is Absolute value of the residual of the ordinary differential equation at the midpoint

$$|\Delta\delta_j| = |\sum_{l=1}^{m+1} \sum_{i=1}^{n_x} e_i \left(a_l(\frac{x_j + x_{j+1}}{2}) B^{(m+1-l)}_{i,k}(\frac{x_j + x_{j+1}}{2}) \right) + a_{m+2}(\frac{x_j + x_{j+1}}{2})|.$$

Let ε_r be the required relative precision and ε_a be the required absolute precision. Consider that the sequence has converged the following condition (2.54) was satisfied.

$$\gamma_j \leq \varepsilon_r \text{ or } |\Delta\delta_j| \leq \varepsilon_a \quad (j = 1, 2, \cdots, n_x - 1) \tag{2.54}$$

If this condition is not satisfied, add the midpoint for which condition (2.54) was not satisfied as a new selected point to the previously used selected points, return to step (b), and perform the calculations again.

(3) **Coefficient determination method**
Let the differential equation to be solved be expressed as:

$$y'' + a_1(x)y' + a_2(x)y + a_3(x) = 0 \tag{2.55}$$

Use an unknown coefficient c to let the solution of (2.55) be expressed as:

$$y(x) = y_1(x) + cy_2(x) \tag{2.56}$$

Differentiate both sides to obtain:

$$y' \quad = \quad y'_1(x) + cy'_2(x) \tag{2.57}$$

$$y'' \quad = \quad y''_1(x) + cy''_2(x) \tag{2.58}$$

By substituting these in (2.55) and rearranging terms, we see that y_1 and y_2 that satisfy the following equations (2.59) and (2.60) will satisfy equations (2.55) and (2.56).

$$y_1'' + a_1(x)y_1' + a_2(x)y_1 + a_3(x) = 0 \tag{2.59}$$
$$y_2'' + a_1(x)y_2' + a_2(x)y_2 = 0 \tag{2.60}$$

Therefore, we will consider solving equations (2.59) and (2.60) here instead of equation (2.55). Boundary conditions can be divided into the following four cases for the left-hand side boundary $x = a$ and the right-hand side boundary $x = b$:

$$y(a) = ya_0 \quad , \quad y(b) = yb_0 \tag{2.61}$$
$$y(a) = ya_0 \quad , \quad y'(b) = yb_1 \tag{2.62}$$
$$y'(a) = ya_1 \quad , \quad y(b) = yb_0 \tag{2.63}$$
$$y'(a) = ya_1 \quad , \quad y'(b) = yb_1 \tag{2.64}$$

where, ya_0, ya_1, yb_0, yb_1 are constants. The calculation procedure for obtaining the unknown coefficient c is shown below.

(a) Initial value problem calculation of (2.59)
For boundary conditions (2.61) and (2.62), if we solve equation (2.59) with initial values:

$$y_2(a) = 0 \quad , \quad y_2'(a) = 1$$

$y_2(b)$ and $y_2'(b)$ are obtained.
For boundary conditions (2.63) and (2.64), if we solve equation (2.59) with initial values:

$$y_2(a) = 1 \quad , \quad y_2'(a) = 0$$

$y_2(b)$ and $y_2'(b)$ are obtained.

(b) Initial value problem calculation of (2.60)
For boundary conditions (2.61) and (2.62), from equation (2.56):

$$\begin{aligned} y_1(a) &= y(a) - cy_2(a) \\ &= ya_0 \end{aligned}$$

Therefore, if we solve equation (2.60) with initial values:

$$y_1(a) = ya_0 \quad , \quad y_1'(a) = 0$$

$y_1(b)$ and $y_1'(b)$ are obtained.
For boundary conditions (2.63) and (2.64), from equation (2.57):

$$\begin{aligned} y_1'(a) &= y'(a) - cy_2'(a) \\ &= ya_1 \end{aligned}$$

Therefore, if we solve equation (2.60) with initial values:

$$y_1(a) = 0 \quad , \quad y_1'(a) = ya_1$$

$y_1(b)$ and $y_1'(b)$ are obtained.

(c) Unknown coefficient c calculation

For boundary conditions (2.61) and (2.63), from equation (2.56):

$$c = \frac{yb_0 - y_1(b)}{y_2(b)}$$

For boundary conditions (2.62) and (2.64), from equation (2.57):

$$c = \frac{yb_1 - y_1'(b)}{y_2'(b)}$$

Using the value of c obtained from the above, obtain $y(x), y'(x)$ and $y''(x)$ from equations (2.56), (2.57) and (2.58).

The Runge-Kutta-Verner method is used for the initial value problem calculation. (See Section 2.1.2) If $y_2(b)$ or $y_2'(b)$ is zero, create an initial value problem facing towards the left-hand side boundary from the the right-hand side boundary and obtain the unknown coefficient c from it. If the denominator is still zero, an error occurs.

2.1.2.3 Integral Equations

(1) **Fredholm's integral equation of the second kind**

Fredholm's integral equation of the second kind is represented by the following expression.

$$y(t) - \int_a^b K(t,x)y(x)dx = f(t)$$

where,

$f(t)$	:	A continuous known function on domain$[a,b]$ of t
$K(t,x)$	:	Kernel (a continuous known function on domain $[a,b]$ of t and x)
$y(t)$	:	Unknown function to be obtained

The solution method when the derivatives of kernel K relative to t and x are continuous (kernel K is a regular kernel) is explained below. Using numerical integration to approximate the second term on the left side of the equation results in

$$\int_a^b K(t,x)y(x)dx \simeq \sum_{i=1}^{n} W_i K(t,x_i)y(x_i)$$

where, W_i, x_i and n are constants determined by the numerical integration formula. Since Gauss' integral formula (30-point formula) is used here, we have

i		W_i	X_i
1	30	0.00796819249616661	0.996893484074650
2	29	0.0184664683110910	0.983668123279747
3	28	0.0287847078833234	0.960021864968308
4	27	0.0387991925696270	0.926200047429274
5	26	0.0484026728305941	0.882560535792053
6	25	0.0574931562176191	0.829565762382768
7	24	0.0659742298821805	0.767777432104826
8	23	0.0737559747377052	0.697850494793316
9	22	0.0807558952294202	0.620526182989243
10	21	0.0868997872010830	0.536624148142020
11	20	0.0921225222377861	0.447033769538089
12	19	0.0963687371746443	0.352704725530878
13	18	0.0995934205867953	0.254636926167890
14	17	0.101762389748406	0.153869913608584
15	16	0.102852652893559	0.051471842555318

(where, a minus sign is attached to X_i for $i = 1, \cdots, 15$)

$$\begin{aligned} x_i &= \frac{|\, a-b \,|}{2} X_i + \frac{a+b}{2} \\ n &= 30 \end{aligned}$$

Using this approximation, Fredholm's integral equation of the second kind is represented as

$$\begin{bmatrix} 1 - W_1K(x_1, x_1) & -W_2K(x_1, x_2) & \cdots & -W_{30}K(x_1, x_{30}) \\ W_1K(x_2, x_1) & 1 - W_2K(x_2, x_2) & \cdots & -W_{30}K(x_2, x_{30}) \\ \vdots & \vdots & \ddots & \vdots \\ W_1K(x_{30}, x_1) & 1 - W_2K(x_{30}, x_2) & \cdots & 1 - W_{30}K(x_{30}, x_{30}) \end{bmatrix} \begin{bmatrix} y(x_1) \\ y(x_2) \\ \vdots \\ y(x_{30}) \end{bmatrix}$$
$$= \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{30}) \end{bmatrix}$$

These simultaneous linear equations of degree 30 are solved using Gauss' elimination method. By using the values $y(x_1), \cdots, y(x_{30})$ that are obtained, $y(t)$ can be obtained at an arbitrary value of t by interpolation using a cubic spline function. (See Section 6.1.2)

(2) **Volterra's integral equation of the first kind**

Volterra's integral equation of the first kind is represented by the following expression.

$$f(t) = \int_a^t K(t, x) y(x) dx$$

where,

$f(t)$: A finite continuous known function on domain $[a, \infty)$ of t

$K(t, x)$: Kernel (a finite continuous known function or a known function that becomes infinitely large at $x = \alpha$ on domain $[a, \infty]$ of t and x)

$$\int_a^x K(x,y)dy < \infty$$

$y(t)$: Unknown function to be obtained

The solution method when the derivatives of kernel K relative to t and x are continuous (kernel K is a regular kernel) is explained below. Applying Maclaurin's formula to the integration interval $[a, a+h]$, we get

$$f(a+h) = \int_a^{a+h} K(a+h,x)y(x)dx = K(a+h, a+\frac{h}{2})y(a+\frac{h}{2})h$$

and $y(a+\frac{h}{2})$ is given by

$$y(a+\frac{h}{2}) \simeq \frac{f(a+h)}{K(a+h, a+\frac{h}{2})h}$$

Applying Maclaurin's formula to the integration intervals $[a, a+2h], [a, a+3h], \cdots, [a, a+6h]$, we obtain the following simultaneous linear equations.

$$\begin{aligned}
f(a+2h) &\simeq 2h\left\{K(a+2h, a+\frac{h}{2})y(a+\frac{h}{2})\right. \\
&\quad \left.+K(a+2h, a+\frac{3}{2}h)y(a+\frac{3}{2}h)\right\} \\
f(a+3h) &\simeq 3h\left\{\frac{3}{8}K(a+3h, a+\frac{h}{2})y(a+\frac{h}{2})\right. \\
&\quad +\frac{2}{8}K(a+3h, a+\frac{3}{2}h)y(a+\frac{3}{2}h) \\
&\quad \left.+\frac{3}{8}K(a+3h, a+\frac{5}{2}h)y(a+\frac{5}{2}h)\right\} \\
f(a+4h) &\simeq 4h\left\{\frac{13}{48}K(a+4h, a+\frac{h}{2})y(a+\frac{h}{2})\right. \\
&\quad +\frac{11}{48}K(a+4h, a+\frac{3}{2}h)y(a+\frac{3}{2}h) \\
&\quad +\frac{11}{48}K(a+4h, a+\frac{5}{2}h)y(a+\frac{5}{2}h) \\
&\quad \left.+\frac{13}{48}K(a+5h, a+\frac{7}{2}h)y(a+\frac{7}{2}h)\right\} \\
f(a+5h) &\simeq 5h\left\{\frac{275}{1152}K(a+5h, a+\frac{h}{2})y(a+\frac{h}{2})\right. \\
&\quad +\frac{100}{1152}K(a+5h, a+\frac{3}{2}h)y(a+\frac{3}{2}h) \\
&\quad +\frac{402}{1152}K(a+5h, a+\frac{5}{2}h)y(a+\frac{5}{2}h) \\
&\quad +\frac{100}{1152}K(a+5h, a+\frac{7}{2}h)y(a+\frac{7}{2}h) \\
&\quad \left.+\frac{275}{1152}K(a+5h, a+\frac{9}{2}h)y(a+\frac{9}{2}h)\right\} \\
f(a+6h) &\simeq 6h\left\{\frac{247}{1280}K(a+6h, a+\frac{h}{2})y(a+\frac{h}{2})\right. \\
&\quad +\frac{139}{1280}K(a+6h, a+\frac{3}{2}h)y(a+\frac{3}{2}h)
\end{aligned}$$

$$
\begin{aligned}
&+\frac{254}{1280}K(a+6h, a+\frac{5}{2}h)y(a+\frac{5}{2}h) \\
&+\frac{254}{1280}K(a+6h, a+\frac{7}{2}h)y(a+\frac{7}{2}h) \\
&+\frac{139}{1280}K(a+6h, a+\frac{9}{2}h)y(a+\frac{9}{2}h) \\
&+\frac{247}{1280}K(a+6h, a+\frac{11}{2}h)y(a+\frac{11}{2}h)\Big\}
\end{aligned}
$$

Solving this, we obtain, $y(a+\frac{3}{2}h), y(a+\frac{5}{2}h), y(a+\frac{7}{2}h), y(a+\frac{9}{2}h)$ and $y(a+\frac{11}{2}h)$.
Next, by solving similar simultaneous linear equations for the integration intervals $[a+5h, a+7h]$, $[a+5h, a+8h], \cdots, [a+5h, a+11h]$, we can obtain $y(a+\frac{13}{2}h), y(a+\frac{15}{2}h), \cdots, y(a+\frac{21}{2}h)$.

In a similar manner, we can obtain the value at each point $y(a+\frac{2j-1}{2}h)$ for $(j=1, \cdots, n)$. By using the values of y that are obtained, $y(x)$ can be obtained at an arbitrary value of x by interpolation using a cubic spline function. (See Section 6.1.2)

2.1.2.4 Partial Differential Equations

(1) **Finite-Difference Approximation**

This library uses the finite difference to solve the following partial differential equations.

$$\nabla^2 + \lambda u = f \tag{2.65}$$

The partial derivatives of the function of two variables on a two-dimensional space $u(x, y)$ and the function of three variables on a three-dimensional space $u(x, y, z)$ are assumed to be sufficiently differentiable and continuous. Also, u_{xx} and u_{yy} are second order partial derivatives with respect to x and to y, respectively, of the function of two variables $u(x, y)$.

$$u(x+h, y) = u(x, y) + u_x(x, y)h + u_{xx}(x, y)\frac{h^2}{2!} + \cdots$$

$$u(x-h, y) = u(x, y) - u_x(x, y)h + u_{xx}(x, y)\frac{h^2}{2!} - \cdots$$

Similarly, if the functions of two variables $u(x, y+k)$ and $u(x, y-k)$ are each expanded as Taylor series for y, the following equations are obtained.

$$u(x, y+k) = u(x, y) + u_y(x, y)h + u_{yy}(x, y)\frac{k^2}{2!} + \cdots$$

$$u(x, y-k) = u(x, y) - u_y(x, y)h + u_{yy}(x, y)\frac{k^2}{2!} - \cdots$$

From the above equations, the central differences for $u_{xx}(x, y)$ and $u_{yy}(x, y)$ are obtained.

$$u_{xx}(x, y) \cong \frac{1}{h^2}[u(x+h, y) - 2u(x, y) + u(x-h, y)]$$

$$u_{yy}(x, y) \cong \frac{1}{k^2}[u(x, y+k) - 2u(x, y) + u(x, y+k)]$$

Using these central differences, equation (2.65) will be as follows.

$$\frac{1}{h^2}[\ u(x+h, y) - 2u(x, y) + u(x-h, y)\ ] +$$

$$\frac{1}{k^2}[\ u(x,y+k) - 2u(x,y) + u(x,y+k)\ ] + \lambda\ u(x,y) = \ f(x,y) \tag{2.66}$$

Similarly, for three dimensions, equation (2.67) is derived by expanding $u(x+h,y,z)$, $u(x-h,y,z)$, $u(x,y+k,z)$, $u(x,y-k,z)$, $u(x,y,z-l)$, and $u(x,y,z+l)$ as Taylor series for x, y, and z, respectively, obtaining the central differences, and substituting them in equation (2.65).

$$\frac{1}{h^2}[\ u(x+h,y,z) - 2u(x,y,z) + u(x-h,y,z)\ ]+$$

$$\frac{1}{k^2}[\ u(x,y+k,z) - 2u(x,y,z) + u(x,y+k,z)\ ]+$$

$$\frac{1}{l^2}[\ u(x,y,z+l) - 2u(x,y,z) + u(x,y,z+l)\ ] + \lambda\ u(x,y,z) = \ f(x,y,z) \tag{2.67}$$

Calculating equation (2.66) or (2.67) at each grid point within a discretized region produces simultaneous linear equations with the value of u at each grid point as the variable, and the value of u is obtained by solving those simultaneous linear equations.

(2) **Finite-Difference Approximation of Boundary Condition**

This section explains the handling of boundary conditions for Dirichlet problems and Neumann problems in this library. In both cases, a virtual grid is established outside of the given area, and the boundary conditions are applied on that virtual grid. The Dirichlet condition assigns the u value on the virtual grid, and the Neumann condition assigns the differential coefficient $\frac{\partial u}{\partial n}$ in the direction of the outer-facing normal of u. Within this library, at the first grid point $(i,j) = (1,1)$ of a discretized rectangular area, for example, the boundary conditions of the bottom edge of the area according to equation (2.66) are as follows.

$$\begin{cases} \text{Dirichlet condition} & u(i,0) = S \quad (S : \text{Value of boundary condition u}) \\ \text{Neumann condition} & u(i,1) = u(i,0) \end{cases}$$

2.1.3 Reference Bibliography

(1) Verner, J. H. , "Explicit Runge-Kutta methods with estimate of the local truncation error", SIAM J. Numer. Anal. Vol.15, No.4, pp.618-641, (1978).

(2) Krogh, F. T. , "A Variable Step Variable Order Multistep Method for the Numerical Solution of Ordinary Differential Equations", Information Processing 68, North-Holand Pub. Co. , pp.194-199, (1968).

(3) Kantaris, N. and Howden, P. F. , "The Universal Equation Solver", SIGMA PRESS, (1983).

(4) Gupta, G. K. , Sacks-Davis, R. and Tischer, P. E. , "A Review of Recent Development in Solving ODEs", ACM Comp. Surveys, Vol.17, No.1, pp.5-47, (1985).

(5) Shampine, L. F. and Gordon, M. K. , "Computer Solution of Ordinary Differential Equations", Freeman, (1975).

(6) Shampine, L. F. , Watts, H. A. and Davenport, S. M. , "Solving Nonstiff Ordinary Differential Equations-the State of the Art", SIAM Review, Vol.18, No.3, pp.376-411, (1976).

(7) Byrne, G. D. and Hindmarsh, A. C. , "A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations", ACM Trans. Math. Softw. , Vol.1, pp.71-96, (1975).

(8) 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, (1977).

(9) R. F. Churchhouse, ed. , "Handbook of Applicable Mathematics, vol. III", John Wiley & Sons Inc. , (1981)

2.2 ORDINARY DIFFERENTIAL EQUATIONS (INITIAL VALUE PROBLEMS)

2.2.1 ASL_dksncs, ASL_rksncs
High-Order Simultaneous Ordinary Differential Equations (Speed Priority)

(1) **Function**

ASL_dksncs or ASL_rksncs solves an ordinary differential equation initial value problem based on automatic step size control when function evaluation is inexpensive and precision requirements are not high.

(2) **Usage**

Double precision:

```
ierr = ASL_dksncs  (f, &x, y, n, mx, m, xf, er, ea, &nst, isr, wk);
```

Single precision:

```
ierr = ASL_rksncs  (f, &x, y, n, mx, m, xf, er, ea, &nst, isr, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n) that defines the differential equations as functions of x and y.
2	x	{D∗ / R∗}	1	Input	Initial value x_0 of independent variable x
				Output	Final destination point x_e of independent variable x
3	y	{D∗ / R∗}	See Contents	Input	Initial values of $y_i^{(j)}$ $(i = 1, \cdots, \mathrm{n};\ j = 0, \cdots, \mathrm{m[i-1]} - 1)$ at $x = x_0$. y[(i − 1) + n × j] = $\mathrm{y_i^{(j)}}$ **Size**: n×(mx + 1)
				Output	Calculated solution $y_i^{(j)}$ $(i = 1, \cdots, \mathrm{n};\ j = 0, \cdots, \mathrm{m[i-1]})$ at $x = x_e$.
4	n	I	1	Input	Number of simultaneous equations
5	mx	I	1	Input	Maximum differential order (max(m[i − 1]))
6	m	I*	n	Input	Differential order m[i−1] of the left-hand side of each of the simultaneous equations

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
7	xf	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Final point x_f where solution is to be obtained
8	er	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
9	ea	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local absolute precision (Default value: Expressible positive minimum value $\times 2^{24}$)
10	nst	I*	1	Input	Step count initial value (Enter 0 when the function is called for the first time)
				Output	Total calculated step count (Input value when integrating consecutively)
11	isr	I	1	Input	0: Integration up to x_f is performed and output. If no error occurs, $x_e = x_f$. Non-zero: x_e and $y_i^{(j)}$ are output each time the step size automatically advances between x_0 and x_f. The final output is at the point x_f, with $x_e = x_f$.
12	wk	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Work	Work area The step size that was used last is entered in wk[0]. **Size**: $8 \times \mathrm{n} \times (\mathrm{mx}+1)+1$
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{n} \geq 1, \mathrm{nst} \geq 0$

(b) $\mathrm{m[i-1]} \geq 1, \mathrm{mx} \geq \max(\mathrm{m[i-1]})$ $(\mathrm{i} = 1, \cdots, \mathrm{n})$

(c) er$\geq e_r$ where e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(d) ea $\geq$ (Expressible positive minimum value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (c) or (d) was not satisfied.	Processing is performed with er or ea set to the default value.
3000	Restriction (a) or (b) was not satisfied.	Processing is aborted.
4000	The step size became too small during the calculation.	The value of x_e and $y_i^{(j)}$ at that time are output, and processing is aborted.

(6) **Notes**

(a) The actual name of function f(x, y, n), that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the high-order simultaneous ordinary differential equations:

$$\begin{cases} y_1^{(m_1)} = f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \quad\vdots \quad\vdots \qquad\qquad\qquad\qquad \vdots \\ y_i^{(m_i)} = f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \quad\vdots \quad\vdots \qquad\qquad\qquad\qquad \vdots \\ y_n^{(m_n)} = f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \end{cases}$$

(Where, if left-hand side is $y_i^{(m_i)}$, then differential order j of corresponding right-hand side $y_i^{(j)}$ must satisfy $j \le m_i - 1$.) this function f(x, y, n) (in double-precision) should be created as follows:

void FORTRAN f(double *x, double *y, int *n)
{
$y[(*n) * m[0]] = f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)});$
$\vdots$
$y[i - 1 + (*n) * m[i - 1]] = f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)});$
$\vdots$
$y[(*n) - 1 + (*n) * m[(*n) - 1]] = f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)});$
}

where the following correspondences are assumed.
$x \leftrightarrow$ *x, $n \leftrightarrow$ *n, $y_i \leftrightarrow$ y[i − 1], $y_i^{(j)} \leftrightarrow$ y[i − 1 + (*n) * j]

Example:

$$\begin{cases} 8(y_1'')^2 - 2 - y_2 y_1' = 0 \quad (y_1'' > 0) \cdots\cdots \text{①} \\ (y_2')^2 - 1 - (y_1')^2 = 0 \quad (y_2' > 0) \ \cdots\cdots \text{②} \end{cases}$$

According to equation ①, y_1'' is: $y_1'' = \sqrt{\frac{1}{4} + \frac{1}{8} \cdot y_2 \cdot y_1'}$
According to equation ②, y_2' is: $y_2' = \sqrt{1 + (y_1')^2}$
Therefore, define the function as follows:

void FORTRAN f(double *x, double *y, int *n)
{
y[4] = sqrt(0.25 + 0.125 * y[1] * y[2]);
y[3] = sqrt(1.0 + y[2] * y[2]);
}

The Input arguments: n=2, mx=2, m[0]=2, m[1]=1. Assign initial values for y[0], y[2] and y[1].

(b) Set nst=0 when integrating for the first time.

(c) When integrating continuously, use the output value of x, y, and nst directly as the next input values.

(d) If ierr=4000, you can either use function 2.2.4 $\begin{Bmatrix} \text{ASL\_dkssca} \\ \text{ASL\_rkssca} \end{Bmatrix}$ beginning from point x_e, or you can continue to solve the problem by making the required precision more lenient.

(e) This function uses the Runge-Kutta-Verner method.

(7) **Example**

(a) Problem

Solve the following simultaneous quadratic ordinary differential equations:

$$\begin{cases} y_1'' = -\frac{y_1}{\gamma} \\ y_2'' = -\frac{y_2}{\gamma} \quad \gamma = (y_1^2 + y_2^2)^{\frac{3}{2}} \end{cases}$$

based on the following initial conditions at $x = 0.0$.

$$y_1(0.0) = 1.0, y_1'(0.0) = 0.0, y_2(0.0) = 0.0, y_2'(0.0) = 1.0$$

(b) Input data

Name of function f(x, y, n):f, x=0.0, n=2, y[0]=1.0, y[2]=0.0, y[1]=0.0, y[3]=1.0, mx=2, m[0]=2, m[1]=2, xf, er, ea, nst=0 and isr=0.

(c) Main program

```
/*      C interface example for ASL_dksncs */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n)
#else
void    f(x,y,n)
double *x;
double *y;
int    *n;
#endif
{
    double r;

    r= pow(y[0]*y[0]+y[1]*y[1],1.5);
    y[2*(*n)]  = -y[0]/r;
    y[2*(*n)+1]= -y[1]/r;

}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    double *y;
    int n;
    int mx;
    int *m;
    double xf;
    double epr;
    double epa;
    int nst;
    int isr;
    double *wk;
    int ierr;
    int i,j,k,nwk;
    FILE *fp;

    n =2;
    mx=2;
    nst =0;
    isr =0;
    fp = fopen( "dksncs.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dksncs ***\n" );
    printf( "\n     ** Input **\n\n" );


    m  = ( int * )malloc((size_t)( sizeof(int) * n ));
    if( m  == NULL )
    {
```

```
        printf( "no enough memory for array m \n" );
        return -1;
    }

    y  = ( double * )malloc((size_t)( sizeof(double) * (n*(mx+1)) ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y \n" );
        return -1;
    }

    nwk=8*n*(mx+1)+1;
    wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
        return -1;
    }

    m[0]=2;
    m[1]=2;
    fscanf( fp, "%lf", &x );
    printf( "\tx   =%8.3g\n", x   );
    for( i=0 ; i<n ; i++ )
    {
        for( j=0 ; j<mx ; j++ )
        {
            fscanf( fp, "%lf", &y[i+n*j] );
            printf( "\ty[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
        }
    }

    printf( "\n\tn   =  %6d\n", n );
    printf( "\tmx  =  %6d\n", mx);
    printf( "\tm[0]=  %6d\n", m[0]);
    printf( "\tm[1]=  %6d\n", m[1]);
    fscanf( fp, "%lf", &epr );
    fscanf( fp, "%lf", &epa );
    printf( "\ter  =%8.3g\n", epr );
    printf( "\tea  =%8.3g\n", epa );
    printf( "\tnst =  %6d\n", nst );
    printf( "\tisr =  %6d\n", isr );

    fclose( fp );

    for (k = 3;k <= 6;k += 3)
    {
        xf = (double)k;
        if (k==6) printf( "\n     ** Input **\n\n" );
        printf( "\txf =%8.3g\n", xf   );

        ierr = ASL_dksncs(f, &x, y, n, mx, m, xf, epr, epa, &nst, isr, wk);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tx = %8.3g\n", x );

        printf( "\n\tSolution\n\n" );
        for( i=0 ; i<n ; i++ )
        {
            for( j=0 ; j<(mx+1) ; j++ )
            {
            printf( "\t  y[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
            }
            printf( "\n" );
        }

        printf( "\n\tStep Number of Calculation\n\n" );
        printf( "\t  nst  = %6d\n", nst  );
    }

    free( m  );
    free( y  );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dksncs ***

 ** Input **

x   =       0
y[     0][     0]=       1
y[     0][     1]=       0
y[     1][     0]=       0
```

```
y[     1][     1]=       1

n   =       2
mx  =       2
m[0]=       2
m[1]=       2
er  =       0
ea  =   1e-10
nst =       0
isr =       0
xf =       3

 ** Output **

ierr =      0

x =        3

Solution

  y[     0][     0]=   -0.99
  y[     0][     1]=  -0.141
  y[     0][     2]=    0.99

  y[     1][     0]=   0.141
  y[     1][     1]=   -0.99
  y[     1][     2]=  -0.141


Step Number of Calculation

  nst  =     56

 ** Input **

xf =       6

 ** Output **

ierr =      0

x =        6

Solution

  y[     0][     0]=    0.96
  y[     0][     1]=   0.279
  y[     0][     2]=   -0.96

  y[     1][     0]=  -0.279
  y[     1][     1]=    0.96
  y[     1][     2]=   0.279


Step Number of Calculation

  nst  =    112
```

2.2.2 ASL_dksnca, ASL_rksnca
High-Order Simultaneous Ordinary Differential Equations (Precision Priority)

(1) **Function**

ASL_dksnca or ASL_rksnca solves ordinary differential equations based on automatic step size and automatic order control when function evaluation is expensive and precision requirements are high.

(2) **Usage**

Double precision:

```
ierr = ASL_dksnca  (f, &x, y, n, mx, m, xf, er, ea, &nst, isr, iwk, wk);
```

Single precision:

```
ierr = ASL_rksnca  (f, &x, y, n, mx, m, xf, er, ea, &nst, isr, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n) that defines the differential equations as functions of x and y.
2	x	{D∗ / R∗}	1	Input	Initial value x_0 of independent variable x
				Output	Final destination point x_e of independent variable x
3	y	{D∗ / R∗}	See Contents	Input	Initial values of $y_i^{(j)}(i = 1, \cdots, \mathrm{n};\ j = 0, \cdots, \mathrm{m[i-1]}-1)$ at $x = x_0$. $\mathrm{y[(i-1)+n\times j]} = \mathrm{y_i^{(j)}}$ **Size**: $\mathrm{n}\times(\mathrm{mx}+1)$
				Output	Calculated solution $y_i^{(j)}(i = 1, \cdots, \mathrm{n};\ j = 0, \cdots, \mathrm{m[i-1]})$ at $x = x_e$.
4	n	I	1	Input	Number of simultaneous equations
5	mx	I	1	Input	Maximum differential order (max(m[i − 1]))
6	m	I*	n	Input	Differential order m[i−1] of the left-hand side of each of the simultaneous equations
7	xf	{D / R}	1	Input	Final point x_f where solution is to be obtained

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	er	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local relative precision **Default value**: Double precision : 10^{-14} Single precision : 10^{-5}
9	ea	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local absolute precision (Default value: Expressible positive minimum value $\times 2^{24}$)
10	nst	I*	1	Input	Step count initial value (Enter 0 when the function is called for the first time)
				Output	Total calculated step count (Input value when integrating consecutively)
11	isr	I	1	Input	0: Integration up to x_f is performed and output. If no error occurs, $x_e = x_f$. Non-zero: x_e and $y_i^{(j)}$ are output each time the step size automatically advances between x_0 and x_f. The final output is at the point x_f, with $x_e = x_f$.
12	iwk	I*	$2 \times \mathrm{n} + 3$	Work	Work area The order q that was used in equation i is entered in iwk[$i-1$].
13	wk	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Work	Work area The step size that was used last is entered in wk[0]. **Size**: $\mathrm{mx} \times (\mathrm{n}+19) + 20 \times \mathrm{n} + 40$
14	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n≥1, nst≥0

(b) $\mathrm{m[i-1]} \geq 1, \mathrm{mx} \geq \max(\mathrm{m[i-1]})$ $(\mathrm{i} = 1, \cdots, \mathrm{n})$

(c) er≥ e_r. Where, e_r= double precision:10^{-14}, single precision:10^{-5}
(Except when 0.0 is entered in order to set er to the default value)

(d) ea ≥ (Expressible positive minimum value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (c) or (d) was not satisfied.	Processing is performed with er or ea set to the default value.
3000	Restriction (a) or (b) was not satisfied.	Processing is aborted.
4000	The step size became too small during the calculation.	The value of x_e and $y_i^{(j)}$ at that time are output, and processing is aborted.

(6) **Notes**

(a) The actual name of function f(x, y, n), that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the high-order simultaneous ordinary differential equations:

$$\begin{cases} y_1^{(m_1)} &= f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \vdots & \vdots \qquad\qquad \vdots \\ y_i^{(m_i)} &= f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \vdots & \vdots \qquad\qquad \vdots \\ y_n^{(m_n)} &= f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \end{cases}$$

(Where, if left-hand side is $y_i^{(m_i)}$, then differential order j of corresponding right-hand side $y_i^{(j)}$ must satisfy $j \leq m_i - 1$.) this function f(x, y, n) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n)
{
  y[(*n) * m[0]] = f_1(x, y_1, ..., y_i, ..., y_i^(j), ..., y_i^(m_i-1), ..., y_n^(m_n-1));
    ⋮
  y[i - 1 + (*n) * m[i - 1]] = f_i(x, y_1, ..., y_i, ..., y_i^(j), ..., y_i^(m_i-1), ..., y_n^(m_n-1));
    ⋮
  y[(*n) - 1 + (*n) * m[(*n) - 1]] = f_n(x, y_1, ..., y_i, ..., y_i^(j), ..., y_i^(m_i-1), ..., y_n^(m_n-1));
}
```

where the following correspondences are assumed.
$x \leftrightarrow$ *x, $n \leftrightarrow$ *n, $y_i \leftrightarrow$ y[i − 1], $y_i^{(j)} \leftrightarrow$ y[i − 1 + (*n) * j]

Example:

$$\begin{cases} 8(y_1'')^2 - 2 - y_2 y_1' = 0 \quad (y_1'' > 0) \cdots\cdots ① \\ (y_2')^2 - 1 - (y_1')^2 = 0 \quad (y_2' > 0) \ \cdots\cdots ② \end{cases}$$

According to equation ①, y_1'' is: $y_1'' = \sqrt{\frac{1}{4} + \frac{1}{8} \cdot y_2 \cdot y_1'}$
According to equation ②, y_2' is: $y_2' = \sqrt{1 + (y_1')^2}$
Therefore, define the function as follows:

```
void FORTRAN f(double *x, double *y, int *n)
{
  y[2 * (*n)] = sqrt(0.25 + 0.125 * y[1] * y[(*n)]);
```

y[1 + (∗n)] = sqrt(1.0 + y[(∗n)] ∗ y[(∗n)]);
}

The Input arguments: n=2, mx=2, m[0]=2, m[1]=1. Assign initial values for y[0], y[2], y[1].

(b) Set nst=0 when integrating for the first time.

(c) When integrating continuously, use the output value of x, y, and nst directly as the next input values. Also, work area iwk and wk must never be overwritten.

(d) For single-precision calculations, since ierr=4000 is likely to be output, you should use double precision as much as possible. If ierr=4000, you can either use function 2.2.4 $\left\{\begin{matrix}\text{ASL\_dkssca}\\ \text{ASL\_rkssca}\end{matrix}\right\}$ beginning from point x_e, or you can continue to solve the problem by making the required precision more lenient.

(e) This function uses a linear multistep method that uses a quotient difference method.

(7) **Example**

(a) Problem

Solve the following simultaneous quadratic ordinary differential equations:

$$\begin{cases} y_1'' = -\frac{y_1}{\gamma} \\ y_2'' = -\frac{y_2}{\gamma} \quad \gamma = (y_1^2 + y_2^2)^{\frac{3}{2}} \end{cases}$$

based on the following initial conditions at $x = 0.0$.

$$y_1(0.0) = 1.0, y_1'(0.0) = 0.0, y_2(0.0) = 0.0, y_2'(0.0) = 1.0$$

(b) Input data

Name of function f(x, y, n):f, x=0.0, n=2, y[0]=1.0, y[2]=0.0, y[1]=0.0, y[3]=1.0, mx=2, m[0]=2, m[1]=2, xf, er, ea, nst=0 and isr=0.

(c) Main program

```
/*      C interface example for ASL_dksnca */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n)
#else
void    f(x,y,n)
double *x;
double *y;
int    *n;
#endif
{
    double r;

    r= pow(y[0]*y[0]+y[1]*y[1],1.5);
    y[2*(*n)]  = -y[0]/r;
    y[2*(*n)+1]= -y[1]/r;

}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    double *y;
    int n;
    int mx;
    int *m;
    double xf;
    double epsr;
```

```
double epsa;
int istep;
int isr;
int *iwk;
double *wk;
int ierr;
int i,j,k,nwk;
FILE *fp;

fp = fopen( "dksnca.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dksnca ***\n" );
printf( "\n     ** Input **\n\n" );
n=2;
mx=2;
istep=0;
isr=0;

m  = ( int * )malloc((size_t)( sizeof(int) * n ));
if( m  == NULL )
{
    printf( "no enough memory for array m \n" );
    return -1;
}

y  = ( double * )malloc((size_t)( sizeof(double) * (n*(mx+1)) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}


iwk  = ( int * )malloc((size_t)( sizeof(int) * (2*n+3) ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk \n" );
    return -1;
}
nwk=mx*(n+19)+20*n+40;
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}

m[0]=2;
m[1]=2;
fscanf( fp, "%lf", &x );
printf( "\tx   =%8.3g\n", x   );
for( i=0 ; i<n ; i++ )
{
    for( j=0 ; j<mx ; j++ )
    {
        fscanf( fp, "%lf", &y[i+n*j] );
        printf( "\ty[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
    }
}

printf( "\n\tn   =  %6d\n", n );
printf( "\tmx  =  %6d\n", mx);
printf( "\tm[0]=  %6d\n", m[0]);
printf( "\tm[1]=  %6d\n", m[1]);
fscanf( fp, "%lf", &epsr );
fscanf( fp, "%lf", &epsa );
printf( "\ter  =%8.3g\n", epsr );
printf( "\tea  =%8.3g\n", epsa );
printf( "\tnst =  %6d\n", istep );
printf( "\tisr =  %6d\n", isr );

fclose( fp );

for (k = 3;k <= 6;k += 3)
{
    xf = (double)k;
    if (k==6) printf( "\n     ** Input **\n\n" );
    printf( "\txf =%8.3g\n", xf   );

    ierr = ASL_dksnca(f, &x, y, n, mx, m, xf, epsr, epsa, &istep, isr, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tx = %8.3g\n", x );
```

```
            printf( "\n\tSolution\n\n" );
            for( i=0 ; i<n ; i++ )
            {
                for( j=0 ; j<(mx+1) ; j++ )
                {
                printf( "\t  y[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
                }
                printf( "\n" );
            }

            printf( "\tStep Number of Calculation\n\n" );
            printf( "\t  nst  = %6d\n", istep  );
        }

        free( y  );
        free( m  );
        free( iwk );
        free(  wk );

        return 0;
    }
```

(d) Output results

```
*** ASL_dksnca ***

 ** Input **

x   =       0
y[     0][     0]=       1
y[     0][     1]=       0
y[     1][     0]=       0
y[     1][     1]=       1

n   =       2
mx  =       2
m[0]=       2
m[1]=       2
er  =       0
ea  =   1e-10
nst =       0
isr =       0
xf =       3

 ** Output **

ierr =      0

x =        3

Solution

  y[     0][     0]=   -0.99
  y[     0][     1]=  -0.141
  y[     0][     2]=    0.99

  y[     1][     0]=   0.141
  y[     1][     1]=   -0.99
  y[     1][     2]=  -0.141

Step Number of Calculation

  nst  =     35

 ** Input **

xf =       6

 ** Output **

ierr =      0

x =        6

Solution

  y[     0][     0]=    0.96
  y[     0][     1]=   0.279
  y[     0][     2]=   -0.96

  y[     1][     0]=  -0.279
  y[     1][     1]=    0.96
  y[     1][     2]=   0.279

Step Number of Calculation

  nst  =     48
```

2.2.3 ASL_dkinct, ASL_rkinct
Implicit Simultaneous Ordinary Differential Equations

(1) **Function**

ASL_dkinct or ASL_rkinct solves an initial value problem when ordinary differential equations are taken as simultaneous equations with implicit ordinary differential equations or with algebraic or nonlinear equations. It also can solve general ordinary differential equations or nonlinear simultaneous equations.

(2) **Usage**

Double precision:

```
ierr = ASL_dkinct  (f, &x, y, n, m, k, xf, idv, &nst, isd, iwk, wk);
```

Single precision:

```
ierr = ASL_rkinct  (f, &x, y, n, m, k, xf, idv, &nst, isd, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of functionf(x, y, n, ji, fn) that defines the differential equations as functions of x and y.
2	x	{ D∗ / R∗ }	1	Input	Initial value x_0 of independent variable x
				Output	Final destination point x_e of independent variable x
3	y	{ D∗ / R∗ }	See Contents	Input	Initial values of $y_i^{(j)}$ at $x = x_0$ $(i = 1, \cdots, \mathrm{n}\ ;\ j = 0, \cdots, \mathrm{m[i-1]} - 1)$. $\mathrm{y[(i-1) + n \times j]} = \mathrm{y_i^{(j)}}$. **Size**: $\mathrm{n}\times(\mathrm{mx}+1)$ Here, $\mathrm{mx} = \max(\mathrm{m[i-1]} + \mathrm{k[i-1]} + \mathrm{isd[i-1]} - 1)$
				Output	Calculated solution $y_i^{(j)}$ at $x = x_e$. $(i = 1, \cdots, \mathrm{n}\ ;\ j = 0, \cdots, \mathrm{m[i-1]})$
4	n	I	1	Input	Number of simultaneous equations
5	m	I*	n	Input	Differential order $\mathrm{m[i-1]}$ of each of the simultaneous equations
6	k	I*	n	Input	Number of equations in the subset consisting of one of the simultaneous equations and the equations obtained by differentiating that equation; $\mathrm{k[i-1]}$

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
7	xf	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Location x_f where solution is to be obtained. (If no error occurs, $x_e = x_f$
8	idv	I	1	Input	Number of integration subdivisions of the interval from x_0 to x_f. (The integration step size becomes $\|x_f - x_0\|$/idv.
9	nst	I*	1	Input	Step count initial value (Enter 0 when the function is used for the first time)
				Output	Total calculated step count (Input value when integrating consecutively)
10	isd	I*	n	Input	Automatic function differentiation switch isd[i − 1] = 0: Automatic differentiation of the i-th equation group is not performed. (k[i − 1] > 1 and the differentiated equations are defined in functionf.) isd[i − 1] = 1: Precision is raised by performing further automatic differentiation of the i-th equation group. (If the i-th equation is difficult to differentiate, then k[i − 1] = 1 and isd[i − 1] = 1 should be set without creating the differentiated equations.)
11	iwk	I*	12 × n + 1	Work	Work area
12	wk	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	24 × n	Work	Work area
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n≥1, nst≥0, idv≥1

(b) $0 \le \mathrm{m[i-1]} \le 4$, $\mathrm{k[i-1]} \ge 1$ $(\mathrm{i} = 1, \cdots, \mathrm{n})$

(c) isd[i − 1]=0 or 1, $\mathrm{m[i-1]} + \mathrm{k[i-1]} + \mathrm{isd[i-1]} \le 6$ $(\mathrm{i} = 1, \cdots, \mathrm{n})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a), (b) or (c) was not satisfied.	Processing is aborted.
4000	Nonlinear simultaneous equations could not be solved.	

(6) **Notes**

(a) The actual name of function f(x, y, n, ji, fn) that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the implicit simultaneous ordinary differential equations:

$$\begin{cases} f_1(x, \cdots, y_1^{(m_1)}, \cdots) = 0 \\ f_2(x, \cdots, y_2^{(m_2)}, \cdots) = 0 \\ \vdots \\ f_i(x, \cdots, y_i^{(m_i)}, \cdots) = 0 \\ \vdots \\ f_n(x, \cdots, y_n^{(m_n)}, \cdots) = 0 \end{cases}$$

(where $y_i^{(m_i)}$ is the term which has maximum differential order and m_i is corresponding differential order), this function f(x, y, n, ji, fn) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n, int *ji, double *fn)
{
  if (*ji==1)
    *fn=f1(x, ..., y1^(m1), ...);
  else if (*ji==2)
    *fn=f1'(x, ..., y1^(m1+1), ...) ;
  else if (*ji==3)
    *fn=f1''(x, ..., y1^(m1+2), ...) ;
    :
  else if (*ji==k1)
    *fn=f1^(k1-1)(x, ..., y1^(m1+k1-1), ...);
  else if (*ji==(k1 + 1))
    *fn=f2(x, ..., y2^(m2), ...);
    :
  else if (*ji==(k1 + k2))
    *fn=f2^(k2-1)(x, ..., y2^(m2+k2-1), ...) ;
    :
  else if (*ji==(sum_{i=1}^{n} ki))
    *fn=fn^(kn-1)(x, ..., yn^(mn+kn-1), ...);
}
```

where the following correspondences are assumed.
$x \leftrightarrow$ *x, $n \leftrightarrow$ *n, $y_i \leftrightarrow$ y[i − 1], $y_i^{(j)} \leftrightarrow$ y[i − 1 + (*n) * j]
In this case, you must carefully determine the number of i-th equation group $k_i \leftrightarrow$ k[i − 1] such that restriction (c) is satisfied. And also you must determine the size of array y from them.

i. If the i-th expression is difficult to differentiate, then either use automatic differentiation by assuming k[i − 1] = 1 and isd[i − 1] = 1 without creating the differentiated equations directly, or use a difference equation. If $f(y'', y', y, x) = 0$ is given, then for $h = \sqrt[3]{\text{Unit for determining error}}$, let

difference equation be as follows:

$$\begin{aligned} f'(y'',y',y,x) = \; & y''' \frac{\partial f(y'',y',y,x)}{\partial y''} + y'' \frac{\partial f(y'',y',y,x)}{\partial y'} \\ & + y' \frac{\partial f(y'',y',y,x)}{\partial y} + \frac{\partial f(y'',y',y,x)}{\partial x} \\ \simeq \; & y''' \frac{f(y''+h,y',y,x) - f(y''-h,y',y,x)}{2h} \\ & + y'' \frac{f(y'',y'+h,y,x) - f(y'',y'-h,y,x)}{2h} \\ & + y' \frac{f(y'',y',y+h,x) - f(y'',y',y-h,x)}{2h} \\ & + \frac{f(y'',y',y,x+h) - f(y'',y',y,x-h)}{2h} \end{aligned}$$

For example, when

$$f(y_1'',y_1',y_1,x) = y_1^2 + xy_1' + \log(\cos(y_1''x^2)) = 0$$

hold and if part of expression $g(y_1'',x) = \log(\cos(y_1''x^2))$ is difficult to differentiate, use the difference equation as follows:

$$\begin{aligned} f'(y_1''',y_1'',y_1',y_1,x) = \; & 2y_1y_1' + y_1' + xy_1'' \\ & + y_1''' \frac{\log(\cos((y_1''+h)x^2)) - \log(\cos((y_1''-h)x^2))}{2h} \\ & + \frac{\log(\cos(y_1''(x+h)^2)) - \log(\cos(y_1''(x-h)^2))}{2h} \end{aligned}$$

Note that $g(y_1'',x)$ is not a function of both y_1' and y_1.

ii. The equations that are taken to be simultaneous should be arranged as follows. A equation, which contains terms of the highest order differential $y_i^{(m_i)} \leftrightarrow$ y[i − 1 + (∗n) ∗ m[i − 1]] of y_i, is a first equation in the i-th equation group. And if plural equations contain terms of the highest order differential $y_i^{(m_i)}$ then it is better to choose a equation that contains the most dominant one for the first equation in the i-th equation group. Further, the first equation in the i-th equation group must contain a term of $y_i^{(m_i)}$.

Example

$$\begin{cases} 3y_1 + 3y_2 + y_3 - 1 = 0 & \cdots\cdots\cdots ① \\ 2y_1' + y_2 + 2y_3 - 6 = 0 & \cdots\cdots\cdots ② \\ y_1 + 2y_2 + 3y_3 - 5 = 0 & \cdots\cdots\cdots ③ \end{cases}$$

These equations should be rearranged so that (②) is a first equation in the first equation group, (①) is a first equation in the second equation group, and (③) is a first equation in the third equation group.

iii. This function also can solve simultaneous nonlinear equations. For this case, the function f(x, y, n, ji, fn) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n, int *ji, double *fn)
{
  if (*ji==1)
    *fn=f_1(x, y_1, ···);
  else if (*ji==2)
    *fn=f_2(x, y_2, ···);
    ⋮
  else if (*ji==(*n))
    *fn=f_n(x, y_n, ···);
}
```

where the following correspondences are assumed.

$x \leftrightarrow *$x, $n \leftrightarrow *$n, $y_i \leftrightarrow$ y[i − 1]

In this usage, let the parameters be as follows:

n = Number of simultaneous equations

m[i − 1] = isd[i − 1] = 0 (i = 1, ⋯ , n) k[i − 1] = 1 (i = 1, ⋯ , n)

x = xf = 0

y[i − 1 + (∗n) ∗ j] (i = 1, ⋯ , n, j = 0) :not necessary (arbitrary).

idv = 1, nst = 0

$y_i = 0 (i = 1, \cdots, n)$ will be assumed as initial values for the calculations.

To solve a nonlinear equation, let n = 1 in the above parameters.

Function creation examples (in double-precision):

i. For an ordinary case:

$$\begin{cases} 8(y_1'')^2 - 2 - y_2 y_1' = 0 \\ (y_2')^2 - 1 - (y_1')^2 = 0 \end{cases}$$

```
void FORTRAN f(double *x, double *y, int *n, int *ji, double *fn)
{
  if (*ji==1)
    *fn = 8.0 * y[(*n) * 2] * y[(*n) * 2] - 2.0 - y[1] * y[(*n)];
    ..... (8(y1'')^2 - 2 - y2 y1' = 0)
  else if (*ji==2)
    *fn = 16.0 * y[(*n) * 2] * y[(*n) * 3] - y[1] * y[(*n) * 2] - y[1 + (*n)] * y[(*n)];
    ..... (16 y1'' y1''' - y2 y1'' - y2' y1' = 0)
  else if (*ji==3)
    *fn = y[1 + (*n)] * y[1 + (*n)] - 1.0 - y[(*n)] * y[(*n)];
    ..... ((y2')^2 - 1 - (y1')^2 = 0)
  else if (*ji==4)
    *fn = 2.0 * y[1 + (*n)] * y[1 + (*n) * 2] - 2.0 * y[(*n)] * y[(*n) * 2]
    ..... (2 y2' y2'' - 2 y1' y1'' = 0)
}
```

Assume the following arguments:

n=2, m[0]=2, m[1]=1, k[0]=2, and k[1]=2 and assign the initial values for y[0], y[2], y[1].

ii. When differential equations are implicitly taken to be simultaneous with an algebraic equation:

$$\begin{cases} y_1^{(3)} - y_2'' - 2y_3' - x^2 = 0 \\ y_2'' - y_3' - \dfrac{x^2}{2} = 0 \\ y_3' - \dfrac{x^2}{2} = 0 \\ y_4 - 2y_1 + y_2 - 2y_3 - 1 = 0 \end{cases}$$

```
void FORTRAN f(double *x, double *y, int *n, int *ji, double *fn)
{
  if (*ji==1)
    *fn = y[(*n) * 3)] - y[1 + (*n) * 2] - 2.0 * y[2 + (*n)] - (*x) * (*x);
    ..... (y1^(3) - y2'' - 2 y3' - x^2 = 0)
  else if (*ji==2)
    *fn = y[(*n) * 4)] - y[1 + (*n) * 3] - 2.0 * y[2 + (*n) * 2] - 2.0 * (*x);
    ..... (y1^(4) - y2^(3) - 2 y3'' - 2x = 0)
```

else if (∗ji==3)
∗ fn = y[(∗n) ∗ 5)] − y[1 + (∗n) ∗ 4] − 2.0 ∗ y[2 + (∗n) ∗ 3] − 2.0;
$\cdots\cdots\cdots (y_1^{(5)} - y_2^{(4)} - 2y_3^{(3)} - 2 = 0)$
else if (∗ji==4)
∗ fn = y[1 + (∗n) ∗ 2)] − y[2 + (∗n)] − (∗x) ∗ (∗x)/2.0;
$\cdots\cdots\cdots (y_2'' - y_3' - \frac{x^2}{2} = 0)$
else if (∗ji==5)
∗ fn = y[1 + (∗n) ∗ 3)] − y[2 + (∗n) ∗ 2] − (∗x);
$\cdots\cdots\cdots (y_2^{(3)} - y_3'' - x = 0)$
else if (∗ji==6)
∗ fn = y[1 + (∗n) ∗ 4)] − y[2 + (∗n) ∗ 3] − 1.0;
$\cdots\cdots\cdots (y_2^{(4)} - y_3^{(3)} - 1 = 0)$
else if (∗ji==7)
∗ fn = y[2 + (∗n)] − (∗x) ∗ (∗x)/2.0;
$\cdots\cdots\cdots (y_3' - \frac{x^2}{2} = 0))$
else if (∗ji==8)
∗ fn = y[2 + (∗n) ∗ 2] − (∗x);
$\cdots\cdots\cdots (y_3'' - x = 0)$
else if (∗ji==9)
∗ fn = y[2 + (∗n) ∗ 3] − 1.0;
$\cdots\cdots\cdots (y_3^{(3)} - 1 = 0)$
else if (∗ji==10)
∗ fn = y[3] − 2.0 ∗ y[0] + y[1] − 2.0 ∗ y[2] − 1.0;
$\cdots\cdots\cdots (y_4 - 2y_1 + y_2 - 2y_3 - 1 = 0)$
}

Assume the following arguments:
n=4, m[0]=3, m[1]=2, m[2]=1, m[3]=0, k[0]=3, k[1]=3, k[2]=3, k[3]=1, isd[i − 1] = 0 (i = $1, \cdots 4$)
and assign initial values for
y[0], y[4], y[8], y[1], y[5], y[2].

iii. For nonlinear simultaneous equations:

$$\begin{cases} (y_1 - 5)^2 + (y_2 - 2)^2 = 4 \\ (y_1 - 2)^2 + (y_2 - 2)^2 = 4 \end{cases}$$

void FORTRAN f(double ∗x, double ∗y, int ∗n, int ∗ji, double ∗fn)
{
if (∗ji==1)
∗ fn = (y[0] − 5.0) ∗ (y[0] − 5.0) + (y[1] − 2.0) ∗ (y[1] − 2.0) − 4.0;
else if (∗ji==2)
∗ fn = (y[0] − 2.0) ∗ (y[0] − 2.0) + (y[1] − 2.0) ∗ (y[1] − 2.0) − 4.0;
}

Assume the following arguments: n=2, m[0]=0, m[1]=0, k[0]=1, k[1]=1, isd[0]=0, isd[1]=0, idv=1, x=0, xf=0 and assume that y[0] and y[1] need not be input. Although these equations have two solution correspond to intersectional points of two circles, a solution closest to the origin $(y_1, y_2) = (0.0, 0.0)$ is returned.

(b) If isd[i − 1] = 1, further automatic differentiation will be performed for the i-th equation group and the precision will increase. However, the calculation speed will decrease somewhat. When k[i − 1] = 1,

isd[i − 1] = 1 should be set except when solving simultaneous nonlinear equations.

(c) If you create as many groups of equations as possible and either increase the number of expressions k[i − 1] in the subsets or increase the number of integration subdivisions idv, then the precision will increase but the calculation speed will decrease.

(d) Set nst = 0 when integrating for the first time.

(e) When integrating continuously, use the output values of x, y, and nst directly as the next input values. Also, work areas iwk and wk must never be overwritten.

(f) Since ierr = 4000 is likely to be output for single-precision calculations, you should use double precision as much as possible. If ierr = 4000, you often will be able to solve the problem by increasing idv.

(7) **Example**

(a) Problem

Solve the following simultaneous third order ordinary differential equations and algebraic equation:

$$\begin{cases} y_1^{(3)} - y_2'' - 2y_3' - x^2 = 0 \\ y_2'' - y_3' - \dfrac{x^2}{2} = 0 \\ y_3' - \dfrac{x^2}{2} = 0 \\ y_4 - 2y_1 + y_2 - 2y_3 - 1 = 0 \end{cases}$$

based on the following initial conditions at $x = 1.0$:
$y_1 = 0.05$, $y_1' = 0.25$, $y_1'' = 1.0$
$y_2 = 0.08333$, $y_2' = 0.333$
$y_3 = 0.1666$

(b) Input data

Name of function f(x, y, n, ji, fn): f, x=0.0, n=4,
y[0]=0.05, y[4]=0.25, y[8]=1.0, y[1]=0.08333, y[5]=0.333, y[2]=0.1666,
m[0]=3, m[1]=2, m[2]=1, m[3]=0,
k[0]=2, k[1]=2, k[2]=2, k[3]=1,
xf=1.5, idv=10, nst=0 (for first use),
isd[0]=1, isd[1]=1, isd[2]=1 and isd[3]=0.

(c) Main program

```
/*      C interface example for ASL_dkinct */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n,int *ji,double *fn)
#else
void    f(x,y,n,ji,fn)
double *x;
double *y;
int    *n;
int    *ji;
double *fn;
#endif
{
    if( *ji == 1 )
    {
        *fn=y[3*(*n)]-y[2*(*n)+1]-2.0*y[(*n)+2]-((*x)*(*x));
    }
    else if( *ji == 2 )
    {
```

```
        *fn=y[4*(*n)]-y[3*(*n)+1]-2.0*y[2*(*n)+2]-((*x)+(*x));
    }
    else if( *ji == 3 )
    {
        *fn=y[2*(*n)+1]-y[(*n)+2]-0.5*((*x)*(*x));
    }
    else if( *ji == 4 )
    {
        *fn=y[3*(*n)+1]-y[2*(*n)+2]-(*x);
    }
    else if( *ji == 5 )
    {
        *fn=y[(*n)+2]-0.5*((*x)*(*x));
    }
    else if( *ji == 6 )
    {
        *fn=y[2*(*n)+2]-(*x);
    }
    else if( *ji == 7 )
    {
        *fn=y[3]-2.0*y[0]+y[1]-2.0*y[2]-1.0;
    }
    else
    {
        printf("error from fkinct ji=%6d\n",*ji);
    }
}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    double *y;
    int n;
    int *m;
    int *k;
    double xf;
    int idv;
    int nst;
    int *isd;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,mx;
    FILE *fp;

    fp = fopen( "dkinct.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dkinct ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=4;
    nst=0;

    m  = ( int * )malloc((size_t)( sizeof(int) * n ));
    if( m  == NULL )
    {
        printf( "no enough memory for array m \n" );
        return -1;
    }

    k  = ( int * )malloc((size_t)( sizeof(int) * n ));
    if( k  == NULL )
    {
        printf( "no enough memory for array  k\n" );
        return -1;
    }

    isd  = ( int * )malloc((size_t)( sizeof(int) * n ));
    if( isd  == NULL )
    {
        printf( "no enough memory for array  isd\n" );
        return -1;
    }
    m[0]=3;
    m[1]=2;
    m[2]=1;
    m[3]=0;
    k[0]=2;
    k[1]=2;
    k[2]=2;
    k[3]=1;
    isd[0]=1;
    isd[1]=1;
    isd[2]=1;
```

```
isd[3]=0;
mx=0;
for( i=0 ; i<n; i++  )
{
    mx=( (mx>m[i]+k[i]+isd[i]-1) ? mx : m[i]+k[i]+isd[i]-1 );
}

y  = ( double * )malloc((size_t)( sizeof(double) * (n*(mx+1)) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}


iwk  = ( int * )malloc((size_t)( sizeof(int) * (12*n+1) ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk \n" );
    return -1;
}
wk  = ( double * )malloc((size_t)( sizeof(double) * (24*n) ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}

fscanf( fp, "%lf", &x );
printf( "\tx   =%8.3g\n", x   );
for( i=0 ; i<n; i++  )
{
    for( j=0 ; j<mx ; j++ )
    {
        y[i+n*j]=0.0;
    }
}
for( i=0 ; i<n;  )
{
    for( j=0 ; j<m[i] ; j++ )
    {
        fscanf( fp, "%lf", &y[i+n*j] );
        printf( "\ty[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
    }
    i++;
}
fscanf( fp, "%lf", &xf);
fscanf( fp, "%d", &idv);

printf( "\n" );
printf( "\tn     =  %6d\n", n );
printf( "\tm[0]  =  %6d\n", m[0]);
printf( "\tm[1]  =  %6d\n", m[1]);
printf( "\tm[2]  =  %6d\n", m[2]);
printf( "\tm[3]  =  %6d\n", m[3]);
printf( "\tk[0]  =  %6d\n", k[0]);
printf( "\tk[1]  =  %6d\n", k[1]);
printf( "\tk[2]  =  %6d\n", k[2]);
printf( "\tk[3]  =  %6d\n", k[3]);
printf( "\txf    =%8.3g\n", xf );
printf( "\tidv   =  %6d\n", idv );
printf( "\tnst   =  %6d\n", nst );
printf( "\tisd[0]=  %6d\n", isd[0]);
printf( "\tisd[1]=  %6d\n", isd[1]);
printf( "\tisd[2]=  %6d\n", isd[2]);
printf( "\tisd[3]=  %6d\n", isd[3]);
printf( "\n" );

fclose( fp );

ierr = ASL_dkinct(f, &x, y, n, m, k, xf, idv, &nst, isd, iwk, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n\tx  = %8.3g\n", x );

printf( "\n\tSolution\n\n" );
for( i=0 ; i<n ; i++ )
{
    for( j=0 ; j<m[i]+1 ; j++ )
    {
        printf( "\ty[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
    }
    printf( "\n" );
}

printf( "\n\tStep Number of Calculation\n\n" );
printf( "\t  nst  = %6d\n", nst  );
```

```
    printf( "\n" );

    free( m  );
    free( k  );
    free( isd );
    free( y  );
    free( wk );
    free( iwk );

    return 0;
}
```

(d) Output results

```
*** ASL_dkinct ***

 ** Input **

x   =       1
y[     0][     0]=    0.05
y[     0][     1]=    0.25
y[     0][     2]=       1
y[     1][     0]=  0.0833
y[     1][     1]=   0.333
y[     2][     0]=   0.167

n     =       4
m[0]  =       3
m[1]  =       2
m[2]  =       1
m[3]  =       0
k[0]  =       2
k[1]  =       2
k[2]  =       2
k[3]  =       1
xf    =     1.5
idv   =      10
nst   =       0
isd[0]=       1
isd[1]=       1
isd[2]=       1
isd[3]=       0


 ** Output **

ierr =      0

x  =      1.5

Solution

y[     0][     0]=    0.38
y[     0][     1]=    1.27
y[     0][     2]=    3.37
y[     0][     3]=    6.75

y[     1][     0]=   0.422
y[     1][     1]=    1.12
y[     1][     2]=    2.25

y[     2][     0]=   0.562
y[     2][     1]=    1.13

y[     3][     0]=    2.46


Step Number of Calculation

  nst  =     10
```

2.2.4 ASL_dkssca, ASL_rkssca
Stiff Problem High-Order Simultaneous Ordinary Differential Equations

(1) **Function**

ASL_dkssca or ASL_rkssca solves a stiff ordinary differential equation initial value problem based on automatic step size and automatic order control.

(2) **Usage**

Double precision:

```
ierr = ASL_dkssca  (f, &x, y, n, mx, m, xf, er, ea, &nst, isr, iwk, wk);
```

Single precision:

```
ierr = ASL_rkssca  (f, &x, y, n, mx, m, xf, er, ea, &nst, isr, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n) that defines the differential equations as functions of x and y.
2	x	{D∗ / R∗}	1	Input	Initial value x_0 of independent variable x
				Output	Final destination point x_e of independent variable x
3	y	{D∗ / R∗}	See Contents	Input	Initial values of $y_i^{(j)}(i = 1, \cdots, \mathrm{n};\ j = 0, \cdots, \mathrm{m[i-1]-1})$ at $x = x_0$. y[(i−1)+n×j] = $\mathrm{y_i^{(j)}}$. **Size**: n×(mx + 1)
				Output	Calculated solution $y_i^{(j)}(i = 1, \cdots, \mathrm{n};\ j = 0, \cdots, \mathrm{m[i-1]})$ at $x = x_e$.
4	n	I	1	Input	Number of simultaneous equations
5	mx	I	1	Input	Maximum differential order (max(m[i − 1]))
6	m	I*	n	Input	Differential order m[i−1] of the left-hand side of each of the simultaneous equations
7	xf	{D / R}	1	Input	Final point x_f where solution is to be obtained

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	er	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local relative precision **Default value**: Double precision : 10^{-14} Single precision : 10^{-5}
9	ea	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local absolute precision (Default value: Expressive positive minimum value $\times 2^{24}$)
10	nst	I*	1	Input	Step count initial value (Enter 0 when the function is called for the first time)
				Output	Total calculated step count (Input value when integrating consecutively)
11	isr	I	1	Input	Parameter that specifies timing for returning to user program. (See Notes (d))
12	iwk	I*	See Contents	Work	Work area **Size**: $\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{m[i-1]} + 6$
13	wk	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Work	Work area The step size that was used last is entered in wk[0]. **Size**: $28 + (\mathrm{k}+9) \times \mathrm{k} + 2 \times \mathrm{n} \times (\mathrm{mx}+1)$ Where, $\mathrm{k} = \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{m[i-1]}$
14	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{n} > 0, \mathrm{nst} \geq 0$

(b) $\mathrm{mx} \geq \mathrm{m[i-1]} \geq 1$ $(i = 1, \cdots, \mathrm{n})$

(c) $\mathrm{isr} == \{0, 1, 2\}$

(d) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(e) ea $\geq$ (Expressible positive minimum value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (d) or (e) was not satisfied.	Processing is performed with er or ea set to the default value.
3000	Restriction (a), (b) or (c) was not satisfied.	Processing is aborted.
4000	The step size became too small during the calculation.	The values of x_e and $y_i^{(j)}$ at that time are output, and processing is aborted.
4100	The equation for obtaining the corrector could not be solved.	The values of x_e and $y_i^{(j)}$ at that time are output, and processing is aborted.

(6) **Notes**

(a) The actual name of function f(x, y, n), that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the high-order simultaneous ordinary differential equations:

$$\begin{cases} y_1^{(m_1)} &= f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \quad\vdots & \quad\vdots \qquad\qquad\qquad\qquad \vdots \\ y_i^{(m_i)} &= f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \quad\vdots & \quad\vdots \qquad\qquad\qquad\qquad \vdots \\ y_n^{(m_n)} &= f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \end{cases}$$

(If the left-hand side is $y_i^{(m_i)}$, then differential order j of corresponding right-hand side $y_i^{(j)}$ must satisfy $j \leq m_i - 1$.), this function f(x, y, n) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n)
{
```

$$\mathrm{y}[(*\mathrm{n}) * \mathrm{m}[0]] = f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)});$$

$$\vdots$$

$$\mathrm{y}[\mathrm{i} - 1 + (*\mathrm{n}) * \mathrm{m}[\mathrm{i} - 1]] = f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)});$$

$$\vdots$$

$$\mathrm{y}[(*\mathrm{n}) - 1 + (*\mathrm{n}) * \mathrm{m}[(*\mathrm{n}) - 1]] = f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)});$$

```
}
```

where the following correspondences are assumed:
$x \leftrightarrow *\mathrm{x}$, $n \leftrightarrow *\mathrm{n}$, $y_i \leftrightarrow \mathrm{y[i-1]}$, $y_i^{(j)} \leftrightarrow \mathrm{y[i-1+(*n)*j]}$

Example:

$$\begin{cases} 8(y_1'')^2 - 2 - y_2 y_1' = 0 \quad (y_1'' > 0) \cdots\cdots \text{①} \\ (y_2')^2 - 1 - (y_1')^2 = 0 \quad (y_{2'} > 0) \cdots\cdots \text{②} \end{cases}$$

According to equation ①, $y_1'' = \sqrt{\frac{1}{4} + \frac{1}{8} \cdot y_2 \cdot y_1'}$.
According to equation ②, $y_2' = \sqrt{1 + (y_1')^2}$.
Therefore, define the function as follows:

void FORTRAN f(double ∗x, double ∗y, int ∗n)
{
y[2 ∗ (∗n)] = sqrt(0.25 + 0.125 ∗ y[1] ∗ y[(∗n)]);
y[1 + (∗n)] = sqrt(1.0 + y[(∗n)] ∗ y[(∗n)]);
}

The input arguments:
n=2, mx=2, m[0]=2, m[1]=1.
Assign initial values for y[0], y[2] and y[1].

(b) If the value assigned for er or ea is less than the default value, then the default value will be set.

(c) If ler is assumed to be a vector of the local relative errors and lea is assumed to be a vector of the local absolute errors, then the solution is accepted if either $\| \text{ler} \| \leq$ er or $\| \text{lea} \| \leq$ ea is satisfied, where $\| \ \|$ is the Max norm. The relative errors are calculated as the error relative to the maximum value of previous calculations.

(d) Specify the value of isr as follows.

isr=0: After integrating past xf, interpolate at xf.

isr=1: Calculate exactly up to xf, without interpolating.

isr=2: Calculate only one integration interval towards xf.

If the differential coefficients are not defined at a point past xf or if there is a point of discontinuity just beyond xf, then specify isr = 1.

(e) When integrating continuously, the output values of x and y must be directly as the next input values. In addition, iwk and wk must not be overwritten.

(f) If ierr = 4000, you can continue to solve the problem continuously by making the required precision more lenient.

(7) **Example**

(a) Problem
Solve the following simultaneous quadratic ordinary differential equations:

$$\begin{cases} y_1'' = -\frac{y_1}{\gamma} \\ y_2'' = -\frac{y_2}{\gamma} \quad \gamma = (y_1^2 + y_2^2)^{\frac{3}{2}} \end{cases}$$

under the following initial conditions at $x = 0.0$:

$$y_1(0.0) = 1.0, y_1'(0.0) = 0.0, y_2(0.0) = 0.0, y_2'(0.0) = 1.0$$

(b) Input data
Name of function f(x, y, n):f, x=0.0, n=2, y[0]=1.0, y[2]=0.0, y[1]=0.0, y[3]=1.0, mx=2, m[0]=2, m[1]=2, xf, er, ea, nst=0 and isr=0.

(c) Main program

```
/*      C interface example for ASL_dkssca */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n)
```

```
#else
void    f(x,y,n)
double *x;
double *y;
int    *n;
#endif
{
    double r;

    r= pow(y[0]*y[0]+y[1]*y[1],1.5);
    y[2*(*n)]  = -y[0]/r;
    y[2*(*n)+1]= -y[1]/r;

}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    double *y;
    int n;
    int mx;
    int *m;
    double xf;
    double er;
    double ea;
    int nst;
    int isr;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,k,summ,nwk;
    FILE *fp;

    fp = fopen( "dkssca.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dkssca ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=2;
    mx=2;
    nst=0;

    m  = ( int * )malloc((size_t)( sizeof(int) * n ));
    if( m  == NULL )
    {
        printf( "no enough memory for array m \n" );
        return -1;
    }

    y  = ( double * )malloc((size_t)( sizeof(double) * (n*(mx+1)) ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y \n" );
        return -1;
    }

    m[0]=2;
    m[1]=2;
    summ=0;
    for( i=0 ; i<n ; i++ )
        summ+=m[i];

    iwk  = ( int * )malloc((size_t)( sizeof(int) * (summ+6) ));
    if( iwk  == NULL )
    {
        printf( "no enough memory for array iwk \n" );
        return -1;
    }
    nwk=28+(summ+9)*summ+2*n*(mx+1);
    wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
        return -1;
    }

    fscanf( fp, "%d", &isr );
    fscanf( fp, "%lf", &x );
    printf( "\tx =%8.3g\n", x   );
    for( i=0 ; i<n ; i++ )
    {
        for( j=0 ; j<mx ; j++ )
        {
            fscanf( fp, "%lf", &y[i+n*j] );
            printf( "\ty[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
```

```
        }
    }

    printf( "\n\tn   =  %6d\n", n );
    printf( "\tmx  =  %6d\n", mx);
    printf( "\tm[0]=  %6d\n", m[0]);
    printf( "\tm[1]=  %6d\n", m[1]);
    fscanf( fp, "%lf", &er );
    fscanf( fp, "%lf", &ea );
    printf( "\ter  =%8.3g\n", er );
    printf( "\tea  =%8.3g\n", ea );
    printf( "\tnst =  %6d\n", nst );
    printf( "\tisr =  %6d\n", isr );

    fclose( fp );

    for (k = 3;k <= 6;k += 3)
    {
        xf = (double)k;
        if (k==6) printf( "\n     ** Input **\n\n" );
        printf( "\txf  =%8.3g\n", xf   );

        ierr = ASL_dkssca(f, &x, y, n, mx, m, xf, er, ea, &nst, isr, iwk, wk);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tx = %8.3g\n", x );

        printf( "\n\tSolution\n\n" );
        for( i=0 ; i<n ; i++ )
        {
            for( j=0 ; j<(mx+1) ; j++ )
            {
            printf( "\t  y[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
            }
            printf( "\n" );
        }

        printf( "\n\tStep Number of Calculation\n\n" );
        printf( "\t  nst  = %6d\n", nst  );
        printf( "\n" );
    }

    free( m  );
    free( y  );
    free( wk );
    free( iwk );

    return 0;
}
```

(d) Output results

```
*** ASL_dkssca ***

 ** Input **

x =       0
y[     0][     0]=       1
y[     0][     1]=       0
y[     1][     0]=       0
y[     1][     1]=       1

n   =       2
mx  =       2
m[0]=       2
m[1]=       2
er  =       0
ea  =       0
nst =       0
isr =       0
xf  =       3

 ** Output **

ierr =      0

x =        3

Solution

  y[     0][     0]=   -0.99
  y[     0][     1]=  -0.141
  y[     0][     2]=    0.99

  y[     1][     0]=   0.141
  y[     1][     1]=   -0.99
  y[     1][     2]=  -0.141
```

```
Step Number of Calculation
  nst  =    551

 ** Input **
xf  =       6
 ** Output **
ierr =      0
x =        6
Solution
  y[     0][     0]=    0.96
  y[     0][     1]=   0.279
  y[     0][     2]=   -0.96
  y[     1][     0]=  -0.279
  y[     1][     1]=    0.96
  y[     1][     2]=   0.279

Step Number of Calculation
  nst  =   1053
```

2.2.5 ASL_dkfncs, ASL_rkfncs
Simultaneous Ordinary Differential Equations of the 1st Order

(1) **Function**

ASL_dkfncs or ASL_rkfncs solves a simultaneous ordinary differential equation of the 1st order initial value problem that satisfies required local precision based on automatic step size control.

(2) **Usage**

Double precision:

```
ierr = ASL_dkfncs  (f, &x, y, n, xf, er, ea, &nst, wk);
```

Single precision:

```
ierr = ASL_rkfncs  (f, &x, y, n, xf, er, ea, &nst, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n) that defines the differential equations as functions of x and y.
2	x	{D∗ / R∗}	1	Input	Initial value x_0 of independent variable x.
				Output	Final destination point x_e of independent variable x
3	y	{D∗ / R∗}	2×n	Input	Initial values of $y_i(i = 1, \cdots, \mathrm{n})$ at $x = x_0$. $\mathrm{y[i-1]} = \mathrm{y_i}$.
				Output	Calculated solution $y_i^{(j)}(i = 1, \cdots, \mathrm{n}; j = 0, 1)$ at $x = x_e$
4	n	I	1	Input	Number of simultaneous equations
5	xf	{D / R}	1	Input	Final point x_f where solution is to be obtained
6	er	{D / R}	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
7	ea	{D / R}	1	Input	Required local absolute precision (Default value: Expressible positive minimum value $\times 2^{24}$)
8	nst	I*	1	Input	Step count initial value (Enter 0 when the function is called for the first time)
				Output	Total calculated step count (Input value when integrating consecutively)
9	wk	{D∗ / R∗}	$16 \times \mathrm{n} + 1$	Work	Work area The step size that was used last is entered in wk[0].
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{n} \geq 1$, $\mathrm{nst} \geq 0$

(b) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5}
(Except when 0.0 is entered in order to set er to the default value)

(c) ea $\geq$ (Expressible positive minimum value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value).

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (b) or (c) was not satisfied.	Processing is performed with er or ea set to the default value.
3000	Restriction (a) was not satisfied.	Processing is aborted.
4000	The step size became too small during the calculation.	The value of x_e and $y_i^{(j)}$ at that time are output, and processing is aborted.

(6) **Notes**

(a) The actual name of function f(x, y, n), that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the simultaneous ordinary differential equations:

$$\begin{cases} y_1' &=& f_1(x, y_1, \cdots, y_i, \cdots, y_n) \\ \vdots & \vdots & \vdots \\ y_i' &=& f_i(x, y_1, \cdots, y_i, \cdots, y_n) \\ \vdots & \vdots & \vdots \\ y_n' &=& f_n(x, y_1, \cdots, y_i, \cdots, y_n) \end{cases}$$

this function f(x, y, n) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n)
{
    y[(*n)] = f1(x, y1, ..., yi, ..., yn);
      :
    y[i - 1 + (*n)] = fi(x, y1, ..., yi, ..., yn);
      :
    y[2 * (*n) - 1] = fn(x, y1, ..., yi, ..., yn);
}
```

where the following correspondences are assumed:
$x \leftrightarrow$ *x, $n \leftrightarrow$ *n, $y_i \leftrightarrow$ y[i − 1]

Example:

$$\begin{cases} y_1' &=& y_2 & \cdots\cdots ① \\ y_2' &=& \dfrac{4y_1}{x^2} + \dfrac{2y_2}{x} & \cdots\cdots ② \end{cases}$$

Define the function as follows:

```
void FORTRAN f(double *x, double *y, int *n)
{
    y[(*n)] = y[1];
    y[1 + (*n)] = 4.0 * y[0]/((*x) * (*x)) + 2.0 * y[1]/(*x);
}
```

and assume that n=2.

(b) Set nst=0 when integrating for the first time.

(c) When integrating continuously, use the output values of x, y, and nst directly as the next input values to solve the problem while successively assigning values for xf.

(d) If ierr= 4000, you can either use function 2.2.4 $\left\{\begin{matrix}\text{ASL\_dkssca}\\ \text{ASL\_rkssca}\end{matrix}\right\}$ beginning from point x_e or you can continue to solve the problem by making the required precision more lenient.

(e) This function uses the Runge-Kutta-Verner method.

(7) **Example**

(a) Problem

Solve that following simultaneous ordinary differential equations of the 1st order for $x = 3.0$ and $x = 6.0$.

$$\begin{cases} y_1' = y_3 \\ y_2' = y_4 \\ y_3' = -\frac{y_1}{\gamma} \\ y_4' = -\frac{y_2}{\gamma} \quad \gamma = (y_1^2 + y_2^2)^{\frac{3}{2}} \end{cases}$$

under the following initial conditions at $x = 0.0$:

$$y_1 = 1.0, y_2 = 0.0, y_3 = 0.0, y_4 = 1.0$$

(b) Input data

Name of function f(x, y, n):f, x=0.0, n=2, y[0]=1.0, y[1]=0.0, y[2]=0.0, y[3]=1.0, xf, er, ea and nst=0 (for the first use).

(c) Main program

```
/*      C interface example for ASL_dkfncs */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n)
#else
void    f(x,y,n)
double *x;
double *y;
int    *n;
#endif
{
    double r;

    r= pow(y[0]*y[0]+y[1]*y[1],1.5);
    y[(*n)]   =  y[2];
    y[(*n)+1]=  y[3];
    y[(*n)+2]= -y[0]/r;
    y[(*n)+3]= -y[1]/r;

}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    double *y;
    int n;
    double xf;
    double epr;
    double epa;
    int nst;
    double *wk;
    int ierr;
    int i,j,k;
    FILE *fp;
```

```
    fp = fopen( "dkfncs.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dkfncs ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=4;
    nst=0;

    y  = ( double * )malloc((size_t)( sizeof(double) * (2*n) ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y \n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (16*n+1) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
        return -1;
    }

    fscanf( fp, "%lf", &x );
    printf( "\tx =%8.3g\n", x   );
    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
        printf( "\ty[%6d][     0]=%8.3g\n",i,y[i] );
    }

    printf( "\n\tn   =  %6d\n", n );
    fscanf( fp, "%lf", &epr );
    fscanf( fp, "%lf", &epa );
    printf( "\ter  =%8.3g\n", epr );
    printf( "\tea  =%8.3g\n", epa );
    printf( "\tnst =  %6d\n", nst );

    fclose( fp );

    for (k = 3;k <= 6;k += 3)
    {
        xf = (double)k;
        if (k==6) printf( "\n     ** Input **\n\n" );
        printf( "\txf  =%8.3g\n", xf   );

        ierr = ASL_dkfncs(f, &x, y, n, xf, epr, epa, &nst, wk);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tx = %8.3g\n", x );

        printf( "\n\tSolution\n\n" );
        for( i=0 ; i<n ; i++ )
        {
            for( j=0 ; j<2 ; j++ )
            {
                printf( "\t  y[%6d][%6d]=%8.3g\n",i,j,y[i+n*j] );
            }
            printf( "\n" );
        }

        printf( "\n\tStep Number of Calculation\n\n" );
        printf( "\t  nst  = %6d\n", nst  );
    }

    free( y  );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dkfncs ***

 ** Input **

x =       0
y[     0][     0]=       1
y[     1][     0]=       0
y[     2][     0]=       0
y[     3][     0]=       1

n   =       4
```

```
er  =       0
ea  =   1e-10
nst =       0
xf  =       3

 ** Output **

ierr =      0

x =        3

Solution

  y[     0][     0]=   -0.99
  y[     0][     1]=  -0.141

  y[     1][     0]=   0.141
  y[     1][     1]=   -0.99

  y[     2][     0]=  -0.141
  y[     2][     1]=    0.99

  y[     3][     0]=   -0.99
  y[     3][     1]=  -0.141


Step Number of Calculation

  nst  =     56

 ** Input **

xf  =       6

 ** Output **

ierr =      0

x =        6

Solution

  y[     0][     0]=    0.96
  y[     0][     1]=   0.279

  y[     1][     0]=  -0.279
  y[     1][     1]=    0.96

  y[     2][     0]=   0.279
  y[     2][     1]=   -0.96

  y[     3][     0]=    0.96
  y[     3][     1]=   0.279


Step Number of Calculation

  nst  =    112
```

2.2.6 ASL_dkhncs, ASL_rkhncs High-Order Ordinary Differential Equation

(1) **Function**

ASL_dkhncs or ASL_rkhncs solves a single high-order ordinary differential equation initial value problem that satisfies required local precision based on automatic step size control.

(2) **Usage**

Double precision:

```
ierr = ASL_dkhncs  (f, &x, y, m, xf, er, ea, &nst, wk);
```

Single precision:

```
ierr = ASL_rkhncs  (f, &x, y, m, xf, er, ea, &nst, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y) that defines the differential equations as functions of x and y.
2	x	{D∗ / R∗}	1	Input	Initial value x_0 of independent variable x.
				Output	Final destination point x_e of independent variable x
3	y	{D∗ / R∗}	m + 1	Input	Initial values of $y^{(j)}(j = 0, \cdots, \mathrm{m}-1)$ at $x = x_0$. y[j] = y(j).
				Output	Calculated solution $y^{(j)}(j = 0, \cdots, \mathrm{m})$ at $x = x_e$
4	m	I	1	Input	Differential order of the left-hand side of the equation (maximum differential order)
5	xf	{D / R}	1	Input	Final point x_f where solution is to be obtained
6	er	{D / R}	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
7	ea	{D / R}	1	Input	Required local absolute precision (Default value: Expressible positive minimum value $\times 2^{24}$)
8	nst	I*	1	Input	Step count initial value (Enter 0 when the function is called for the first time)
				Output	Total calculated step count (Input value when integrating consecutively)
9	wk	{D∗ / R∗}	8 × m + 9	Work	Work area The step size that was used last is entered in wk[0].
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) m ≥ 1, nst ≥ 0

(b) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5}
(Except when 0.0 is entered in order to set er to the default value)

(c) ea $\geq$ (Expressible positive minimum value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (b) or (c) was not satisfied.	Processing is performed with er or ea set to the default value.
3000	Restriction (a) was not satisfied.	Processing is aborted.
4000	The step size became too small during the calculation.	The value of x_e and $y_i^{(j)}$ at that time are output, and processing is aborted.

(6) **Notes**

(a) The actual name of function f(x, y, n), that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the high-order ordinary differential equation:

$$y^{(m)} = f(x, y, \cdots, y^{(j)}, \cdots, y^{(m-1)})$$

(If the left-hand side is $y^{(m)}$, then differential order j of right-hand side $y^{(j)}$ must satisfy $j \leq m-1$, where m is constant.), this function f(x, y) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *y)
{
   y[m] = f(x, y, ···, y^(j), ···, y^(m-1));
}
```

where the following correspondences are assumed.
$x \leftrightarrow$ *x, $y \leftrightarrow$ y[0], $y^{(j)} \leftrightarrow$ y[j]

Example:

$$y'' = \frac{4y}{x^2} + \frac{2y'}{x}$$

Define the function as follows:

```
void FORTRAN f(double *x, double *y)
{
   y[2] = 4.0 * y[0]/((*x) * (*x)) + 2.0 * y[1]/(*x);
}
```

and assume that m=2.

(b) Set nst=0 when integrating for the first time.

(c) When integrating continuously, use the output values of x, y, and nst directly as the next input values to solve the problem while successively assigning values for xf.

(d) If ierr= 4000, you can either use function 2.2.4 $\left\{\begin{matrix}\text{ASL\_dkssca}\\ \text{ASL\_rkssca}\end{matrix}\right\}$ beginning from point x_e or you can continue to solve the problem by making the required precision more lenient.

(e) This function uses the Runge-Kutta-Verner method.

(7) **Example**

(a) Problem

Solve the following ordinary differential equation for $x = 5.0$ and $x = 10.0$:

$$y'' = \frac{4y}{x^2} + \frac{2y'}{x}$$

under the following initial conditions at $x = 1.0$:

$y = 5.0$, $y' = 3.0$.

(b) Input data

Name of function f(x, y): f, x=1.0, y[0]=5.0, y[1]=3.0, m=2, xf, er, ea and nst=0 (for the first use).

(c) Main program

```
/*      C interface example for ASL_dkhncs */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y)
#else
void    f(x,y)
double *x;
double *y;
#endif
{
    y[2]= 4.0*y[0]/((*x)*(*x))+2.0*y[1]/(*x);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    double *y;
    int m;
    double xf;
    double epr;
    double epa;
    int nst;
    double *wk;
    int ierr;
    int i,k;
    FILE *fp;

    fp = fopen( "dkhncs.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dkhncs ***\n" );
    printf( "\n     ** Input **\n\n" );
    m=2;
    nst=0;

    y  = ( double * )malloc((size_t)( sizeof(double) * (m+1) ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y \n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (8*m+9) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
```

```
        return -1;
    }

    fscanf( fp, "%lf", &x );
    printf( "\tx   =%8.3g\n", x   );
    for( i=0 ; i<m ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
        printf( "\ty[%6d]=%8.3g\n",i,y[i] );
    }

    printf( "\n\tm   =  %6d\n", m );
    fscanf( fp, "%lf", &epr );
    fscanf( fp, "%lf", &epa );
    printf( "\ter  =%8.3g\n", epr );
    printf( "\tea  =%8.3g\n", epa );
    printf( "\tnst =  %6d\n", nst );

    fclose( fp );
    for (k = 5;k <= 10;k += 5)
    {
        xf = (double)k;
        if (k==10) printf( "\n     ** Input **\n\n" );
        printf( "\txf  =%8.3g\n", xf   );

        ierr = ASL_dkhncs(f, &x, y, m, xf, epr, epa, &nst, wk);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tx    = %8.3g\n", x );

        printf( "\n\tSolution\n\n" );
        for( i=0 ; i<m+1 ; i++ )
        {
            printf( "\t  y[%6d]=%8.3g\n",i,y[i] );
        }

        printf( "\n\tStep Number of Calculation\n\n" );
        printf( "\t  nst  = %6d\n", nst  );
    }

    free( y  );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dkhncs ***

 ** Input **

x   =       1
y[     0]=       5
y[     1]=       3

m   =       2
er  =       0
ea  =   1e-10
nst =       0
xf  =       5

 ** Output **

ierr =      0

x    =        5

Solution

  y[     0]=   1e+03
  y[     1]=     800
  y[     2]=     480

Step Number of Calculation

  nst  =    119

 ** Input **

xf  =      10

 ** Output **

ierr =      0

x    =       10

Solution
```

```
  y[     0]= 1.6e+04
  y[     1]= 6.4e+03
  y[     2]=1.92e+03

Step Number of Calculation

  nst  =    176
```

2.2.7 ASL_dkmncn, ASL_rkmncn Ordinary Differential Equation of the Type $M\mathbf{y}'' + C\mathbf{y}' + K\mathbf{y} = \mathbf{p}(x)$

(1) **Function**

ASL_dkmncn or ASL_rkmncn solves a matrix form simultaneous ordinary differential equation of 2nd order initial value problem which is known as the equation of motion:

$$M\boldsymbol{y}'' + C\boldsymbol{y}' + K\boldsymbol{y} = \boldsymbol{p}(x)$$

where M is mass matrix, C is damping matrix, K is stiffness matrix.

(2) **Usage**

Double precision:

```
ierr = ASL_dkmncn  (m, c, k, n, po, p, y, h, sta, &isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rkmncn  (m, c, k, n, po, p, y, h, sta, &isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	m	{D∗ / R∗}	n×n	Input	Mass matrix M
2	c	{D∗ / R∗}	n×n	Input	Damping matrix C
3	k	{D∗ / R∗}	n×n	Input	Stiffness matrix K
4	n	I	1	Input	Number of simultaneous equations n
5	po	{D∗ / R∗}	n	Input	External force $\boldsymbol{p}(x_0)$ at $x = x_0$ (A value must be input when the function is called for the first time)
				Output	Updated by value of $\boldsymbol{p}(x_0 + h)$ (Input value when integrating consecutively)
6	p	{D∗ / R∗}	n	Input	External force $\boldsymbol{p}(x_0 + h)$ at $x = x_0 + h$
7	y	{D∗ / R∗}	n×3	Input	Initial values of $y_i^{(j)}(i = 1, \cdots, \mathrm{n}; j = 0, 1)$ at $x = x_0$. (Input a value when the function is used for the first time.) When integrating consecutively, input the value of y[(i − 1) + n ∗ j] that was output during the previous iteration.
				Output	Calculated solution $y_i^{(j)}(i = 1, \cdots, \mathrm{n}; j = 0, 1, 2)$ at $x = x_0 + h$ (Input value when integrating successively)
8	h	{D / R}	1	Input	Step size h of x
9	sta	{D / R}	1	Input	Constant θ for obtaining a solution with stability (Default value is 1.4)

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	isw	I*	1	Input	Step size modification switch. Let isw be zero when the function is used for the first time. When integrating consecutively without changing h and sta, let isw be the output value when the function was used during the previous iteration. When h or sta is changed, let isw be 2.
				Output	Next input value for integrating consecutively Without changing h and sta
11	iwk	I*	n	Work	Work area
12	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	n × n + n	Work	Work area
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 1

(b) sta ≥ 1.0
(except when 0.0 is entered in order to set sta to the default value)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (b) was not satisfied.	Processing is performed with sta set to the default value.
3000	Restriction (a) was not satisfied.	Processing is aborted.
4000	The simultaneous linear equations could not be solved. Although it was the first integration, isw ≠ 0.	

(6) **Notes**

(a) This function calculate solution at $x = x_0 + h$ when the condition at $x = x_0$ for small enough step size h is given. So, if $x_f - x_0$, which is difference between start point x_0 and end point x_f, is large, set step sizes $h_i (i = 1, \cdots, v)$ for which $x_f = x_0 + \sum_{i=1}^{v} h_i$ met, and you should calculate calling v times this function. And if the external force $\boldsymbol{p}(x)$ varies with respect to x, prepare the values of the external force at each point $\boldsymbol{p}(x_0 + \sum_{i=1}^{j} h_i)$ $(j = 1, \cdots, v)$ and they must input for p sequentially.

(b) When integrating for the first time, set isw=0 and input the external force values $\boldsymbol{p}(x_0)$ and $\boldsymbol{p}(x_0 + h)$ for po and p, respectively. Where, h is step size.

(c) When integrating consecutively, use the output values of po and y directly as the next input values. Also, for isw, use the value output during the previous iteration directly as then next input value if

the step size h and constant sta do not change, and set isw=2 if h or sta does change. Also, never overwrite wk.

(d) Although a stable solution is obtained if the value of sta is greater than or equal to 1.37, the error will become quite large if sta is greater than or equal to 2.0. According to Wilson, sta should be around 1.4.

(e) This function uses the Wilson's θ method.

(7) **Example**

(a) Problem

Solve the following a matrix form simultaneous ordinary differential equation of 2nd order:

$$M\boldsymbol{y}'' + C\boldsymbol{y}' + K\boldsymbol{y} = \boldsymbol{p}(x)$$

based on the following initial conditions at $x = x_0$

$$\boldsymbol{y}|_{x=x_0} = \begin{bmatrix} 100.0 \\ 100.0 \end{bmatrix},\ \boldsymbol{y}'|_{x=x_0} = \begin{bmatrix} 100.0 \\ 100.0 \end{bmatrix}.$$

Mass matrix M, damping matrix C and stiffness matrix K are given as follows.

$$M = \begin{bmatrix} 2000.0 & -1000.0 \\ 2000.0 & -2000.0 \end{bmatrix}$$

$$C = \begin{bmatrix} 300.0 & -600.0 \\ 300.0 & -1200.0 \end{bmatrix}$$

$$K = \begin{bmatrix} 2000.0 & -1500.0 \\ 2000.0 & -3000.0 \end{bmatrix}$$

The external force vector $\boldsymbol{p}(x)$ (constant) is given as follows:

$$\boldsymbol{p}(x) = \begin{bmatrix} 100.0 \\ 100.0 \end{bmatrix}.$$

(b) Input data

Mass matrix M, damping matrix C, stiffness matrix K, n=2,
the external force $\boldsymbol{p}(x)$ (set for po and p),
initial values $\boldsymbol{y}|_{x=x_0}$ and $\boldsymbol{y}'|_{x=x_0}$ at $x = x_0$ (set for array y),
step size h=0.01 (set for h, constant for each step),
sta=0.0 and isw=0 (for the first time).
For the input values shown above, output the values at $x = x_0 + 50h$, and $x = x_0 + 100h$, where, $x_0 = 0.0$.

(c) Main program

```
/*      C interface example for ASL_dkmncn */

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

int main()
{
    double *m;
    double *c;
    double *k;
    int n;
    double *po;
    double *p;
```

```
double *y;
double h;
double sta;
int isw;
int *iwk;
double *wk;
int ierr;
double htmp;
int i,j,l;
FILE *fp;

fp = fopen( "dkmncn.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dkmncn ***\n" );
printf( "\n     ** Input **\n\n" );
n=2;
isw=0;

m  = ( double * )malloc((size_t)( sizeof(double) * (n*n) ));
if( m  == NULL )
{
    printf( "no enough memory for array  m\n" );
    return -1;
}

c  = ( double * )malloc((size_t)( sizeof(double) * (n*n) ));
if( c  == NULL )
{
    printf( "no enough memory for array  c\n" );
    return -1;
}

k  = ( double * )malloc((size_t)( sizeof(double) * (n*n) ));
if( k  == NULL )
{
    printf( "no enough memory for array  k\n" );
    return -1;
}

po  = ( double * )malloc((size_t)( sizeof(double) * n ));
if( po  == NULL )
{
    printf( "no enough memory for array po\n" );
    return -1;
}

p  = ( double * )malloc((size_t)( sizeof(double) * n ));
if( p  == NULL )
{
    printf( "no enough memory for array p\n" );
    return -1;
}

y  = ( double * )malloc((size_t)( sizeof(double) * (3*n) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

wk  = ( double * )malloc((size_t)( sizeof(double) * (n*(n+1)) ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * n ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

printf( "\tm[i][j]\n" );
printf( "\t              j=0      j=1\n" );
for( i=0 ; i<n ; i++ )
{
    printf( "\t i=%6d",i );
    for( j=0 ; j<n ; j++ )
    {
        fscanf( fp, "%lf", &m[i+n*j] );
        printf( " %8.3g",m[i+n*j] );
    }
    printf( "\n" );
}
```

```
printf( "\n\tc[i][j]\n" );
printf( "\t               j=0      j=1\n" );
for( i=0 ; i<n ; i++ )
{
    printf( "\t i=%6d",i );
    for( j=0 ; j<n ; j++ )
    {
        fscanf( fp, "%lf", &c[i+n*j] );
        printf( " %8.3g",c[i+n*j] );
    }
    printf( "\n" );
}

printf( "\n\tk[i][j]\n" );
printf( "\t               j=0      j=1\n" );
for( i=0 ; i<n ; i++ )
{
    printf( "\t i=%6d",i );
    for( j=0 ; j<n ; j++ )
    {
        fscanf( fp, "%lf", &k[i+n*j] );
        printf( " %8.3g",k[i+n*j] );
    }
    printf( "\n" );
}

printf( "\n\tn =%6d\n\n", n );
for( i=0 ; i<n ; i++ )
{
    fscanf( fp, "%lf", &po[i] );
    printf( "\tpo[%6d]=%8.3g\n",i,po[i] );
}

for( i=0 ; i<n ; i++ )
{
    fscanf( fp, "%lf", &p[i] );
    printf( "\tp[%6d] =%8.3g\n",i,p[i] );
}

printf( "\n\ty[i][j]\n" );
printf( "\t               j=0      j=1\n" );
for( i=0 ; i<n ; i++ )
{
    printf( "\t i=%6d",i );
    for( j=0 ; j<2 ; j++ )
    {
        fscanf( fp, "%lf", &y[i+n*j] );
        printf( " %8.3g",y[i+n*j] );
    }
    printf( "\n" );
}

fscanf( fp, "%lf", &h );
fscanf( fp, "%lf", &sta );
htmp=h;
printf( "\n\th   =%8.3g\n", h );
printf( "\tsta =%8.3g\n", sta );
printf( "\tisw =  %6d\n", isw );

fclose( fp );
printf( "\n     ** Output **\n\n" );
for ( l = 1; l <= 100; l++ )
{

    ierr = ASL_dkmncn(m, c, k, n, po, p, y, h, sta, &isw, iwk, wk);

    if(l==50 || l==100)
    {
        printf( "\n\tierr = %6d\n", ierr );
        printf( "\n\tx    =%8.3g\n", htmp   );

        printf( "\n\ty[i][j]\n" );
        printf( "\t               j=0      j=1      j=2\n" );
        for( i=0 ; i<n ; i++ )
        {
            printf( "\t i=%6d",i );
            for( j=0 ; j<3 ; j++ )
            {
                printf( " %8.3g",y[i+n*j] );
            }
            printf( "\n" );
        }
    }
    htmp += h;
}
```

```
    free(  m  );
    free(  c  );
    free(  k  );
    free(  po );
    free(  p  );
    free(  y  );
    free(  iwk );
    free(  wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dkmncn ***

 ** Input **

m[i][j]
                 j=0       j=1
 i=     0    2e+03   -1e+03
 i=     1    2e+03   -2e+03

c[i][j]
                 j=0       j=1
 i=     0      300      -600
 i=     1      300 -1.2e+03

k[i][j]
                 j=0       j=1
 i=     0    2e+03 -1.5e+03
 i=     1    2e+03   -3e+03

n =     2

po[     0]=     100
po[     1]=     100
p[     0] =     100
p[     1] =     100

y[i][j]
                 j=0       j=1
 i=     0      100       100
 i=     1      100       100

h   =    0.01
sta =       0
isw =       0

 ** Output **


ierr =      0

x    =     0.5

y[i][j]
                 j=0       j=1       j=2
 i=     0      134        35      -139
 i=     1      124    -1.65      -185

ierr =      0

x    =       1

y[i][j]
                 j=0       j=1       j=2
 i=     0      134     -33.6      -129
 i=     1      103     -76.4      -109
```

2.3 ORDINARY DIFFERENTIAL EQUATIONS (BOUNDARY VALUE PROBLEMS)

2.3.1 ASL_dosnnv, ASL_rosnnv
High-Order Simultaneous Ordinary Differential Equations (Numerical Boundary Conditions)

(1) **Function**

ASL_dosnnv or ASL_rosnnv uses the multipoint shooting method to solve a boundary value problem for high-order simultaneous ordinary differential equations for which boundary conditions are given as input values.

(2) **Usage**

Double precision:

```
ierr = ASL_dosnnv  (f, xa, xb, in, ib, ic, bn, m, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rosnnv  (f, xa, xb, in, ib, ic, bn, m, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n, alf) that defines the differential equations as functions of x and y.
2	xa	{D / R}	1	Input	x coordinate of left-hand side boundary
3	xb	{D / R}	1	Input	x coordinate of right-hand side boundary
4	in	I*	See Contents	Input	Positions where boundary conditions are set (xa side:0, xb side: Nonzero) **Size:** $\sum_{i=1}^{n} m[i-1]$
5	ib	I*	See Contents	Input	Element numbers i of variable $y_i^{(j)}$ which gives boundary conditions value **Size:** $\sum_{i=1}^{n} m[i-1]$

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
6	ic	I*	See Contents	Input	Differential order j of variable $y_i^{(j)}$ which gives boundary conditions value **Size:** $\sum_{i=1}^{n} m[i-1]$
7	bn	$\begin{Bmatrix} D* \\ R* \end{Bmatrix}$	See Contents	Inputs	Boundary condition values given for variable $y_i^{(j)}$ **Size:** $\sum_{i=1}^{n} m[i-1]$
8	m	I*	n	Input	Maximum differential order of variable y_i in differential equations
9	n	I	1	Input	Number of simultaneous equations.
10	x	$\begin{Bmatrix} D* \\ R* \end{Bmatrix}$	k	Input	x coordinate x_i where approximate solutions are calculated
11	k	I	1	Input	Number of calculation points
12	er	$\begin{Bmatrix} D \\ R \end{Bmatrix}$	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
13	ea	$\begin{Bmatrix} D \\ R \end{Bmatrix}$	1	Input	Required local absolute precision (Default value: Unit for determining error)
14	nx	I*	1	Input	Maximum number of shooting points
				Output	Actual number of shooting points
15	nev	I*	1	Input	Maximum number of iterations (Default value :100)
				Output	Actual number of iterations
16	y	$\begin{Bmatrix} D* \\ R* \end{Bmatrix}$	See Contents	Output	Solutions $y_j^{(v)}(x_i)$ $y[(i-1)+k\times(j-1)+k\times n\times v]$ $= y_j^{(v)}(x_i)$ **Size:** $k\times n\times(\max(m[i-1])+1)$
17	isw	I	1	Input	Parameterization processing switch 0: Parameterization not performed Nonzero: Parameterization performed
18	iwk	I*	See Contents	Work	Work area **Size**: $\sum_{i=1}^{n} m[i-1]$
19	wk	$\begin{Bmatrix} D* \\ R* \end{Bmatrix}$	See Contents	Work	Work area **Size**: $(n\times km+1)^2\times(nx+1)+n\times km\times \max(3\times n\times km, km+15)$ Where, $km=\max(m[i-1])$
20	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa < xb

(b) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5}
(Except when 0.0 is entered in order to set er to the default value)

(c) ea $\geq$ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) nev > 0 (Except when 0 is entered in order to set nev to the default value)

(e) n $\geq$ 1

(f) m[i − 1] $\geq$ 1 (i = 1, 2, $\cdots$, n)

(g) 1 $\leq$ ib[i − 1] $\leq$ n and 0 $\leq$ ic[i − 1] < m[ib[i − 1]] (i = 1, 2, $\cdots$, $\sum_{j=1}^{n}$ m[j − 1])

(h) k $\geq$ 1

(i) xa $\leq$ x[i − 1] $\leq$ xb (i = 1, 2, $\cdots$, k)

(j) nx $\geq$ min(5i + 1, 51)
where i is the minimum integer for which xb − xa $\leq$ i

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where xa $\neq$ xb).	Processing is performed assuming xa and xb are replaced each other.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with the corresponding default value set.
3000	xa=xb	Processing is aborted.
3010	Restriction (e) was not satisfied.	
3020	Restriction (f) was not satisfied.	
3030	Restriction (g) was not satisfied.	
3040	Restriction (h) was not satisfied.	
3050	Restriction (i) was not satisfied.	
3060	Restriction (j) was not satisfied.	
4000	The shooting point could not be added due to server oscillation (large derivative), etc.	
4500	The step size became too small during the initial value problem calculation (existence of singularity, etc).	
5000	The maximum number of shooting points was reached.	
5500	The maximum number of iterations was reached (including cases when solution does not exist or is indefinite).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) In a nonlinear problem for high-order simultaneous ordinary differential equations expressed as follows:

$$\begin{cases} y_1^{(m_1)} &= f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \quad\vdots & \quad\vdots \qquad\qquad\qquad\qquad \vdots \\ y_i^{(m_i)} &= f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \quad\vdots & \quad\vdots \qquad\qquad\qquad\qquad \vdots \\ y_n^{(m_n)} &= f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \end{cases}$$

(Where, if left-hand side is $y_i^{(m_i)}$, then differential order j of corresponding right-hand side $y_i^{(j)}$ must satisfy $j \leq m_i - 1$.) if there is a multiplication or division of two or more of the variables $y_i^{(j)}$ in the terms on the right-hand side of any of these equations such as $y_1 \cdot y_1'$, $\frac{y_2}{y_3'}$, y_4^2 or a function other than a sum or scalar multiple of $y_i^{(j)}$ such as $\sin(y_1)$ or $\mathrm{abs}(y_2')$, parameterize this nonlinear problem by multiplying this portion by alf.

Example :

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - y_1 \cdot y_2 \end{cases}$$

Parameterize this nonlinear problem as follows:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

When the problem has been parameterized, set isw = 1. A linear problem need not be parameterized. In this case, set isw = 0.

(b) The actual name of function f(x, y, n, alf), that defines the differential equations, must be declared in the user program, and the actual function must be created.

The function f(x, y, n, alf) (in double-precision) for the high-order simultaneous ordinary differential equations that were parameterized as described in Note (a) should be created as follows:

void FORTRAN f(double ∗x, double ∗y, int ∗n, double ∗alf)
{
$\quad \mathrm{y}[(*\mathrm{n}) * \mathrm{m}[0]] = f_1(*\mathrm{alf}, x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots);$
$\quad\quad\vdots$
$\quad \mathrm{y}[\mathrm{i} - 1 + (*\mathrm{n}) * \mathrm{m}[\mathrm{i} - 1]] = f_i(*\mathrm{alf}, x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots);$
$\quad\quad\vdots$
$\quad \mathrm{y}[(*\mathrm{n}) - 1 + (*\mathrm{n}) * \mathrm{m}[(*\mathrm{n}) - 1]] = f_n(*\mathrm{alf}, x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots);$
}

where the following correspondences are assumed:
$x \leftrightarrow *\mathrm{x}$, $n \leftrightarrow *\mathrm{n}$, $y_i \leftrightarrow \mathrm{y}[\mathrm{i} - 1]$, $y_i^{(j)} \leftrightarrow \mathrm{y}[\mathrm{i} - 1 + (*\mathrm{n}) * \mathrm{j}]$
$f_i(*\mathrm{alf}, x, \cdots)$ is parameterized one for $f_i(x, \cdots)$.

Example:

When parameterized ordinary differential equations are as follows:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

define the function as follows:

```
void FORTRAN f(double *x, double *y, int *n, double *alf)
{
    y[4] = y[1];
    y[3] = y[2] - (*alf) * (y[0] * y[1]);
}
```

The Input arguments: n=2, m[0]=2, m[1]=1, isw=1.

(c) If the boundary conditions of the high-order simultaneous ordinary differential equations are given by:

$$\text{left-hand side boundary} \quad \begin{cases} y_{a_1}^{(b_1)} = c_1 \\ y_{a_2}^{(b_2)} = c_2 \\ \quad \vdots \\ y_{a_p}^{(b_p)} = c_p \end{cases}$$

$$\text{right-hand side boundary} \quad \begin{cases} y_{a_{p+1}}^{(b_{p+1})} = c_{p+1} \\ y_{a_{p+2}}^{(b_{p+2})} = c_{p+2} \\ \quad \vdots \\ y_{a_q}^{(b_q)} = c_q \end{cases} \qquad q = \sum_{i=1}^{n} m_i$$

set array in, ib, ic, and bn as follows: in[i − 1] = 0 (i = 1, 2, ⋯ , p)
in[i − 1] = 1 (i = p + 1, p + 2, ⋯ , q)
ib[i − 1] = a_i(i = 1, 2, ⋯ , q)
ic[i − 1] = b_i(i = 1, 2, ⋯ , q)
bn[i − 1] = c_i(i = 1, 2, ⋯ , q)

Example: If the boundary conditions are given by:

$$\text{left-hand side boundary} \quad \begin{cases} y_1' = 0.0 \\ y_2 = 1.0 \end{cases} \quad \text{right-hand side boundary} \quad y_1' = 2.0$$

the input values of in, ib, ic, and bn are as follows:
in[0]=0, ib[0]=1, ic[0]=1, bn[0]=0.0
in[1]=0, ib[1]=2, ic[1]=0, bn[1]=1.0
in[2]=1, ib[2]=1, ic[2]=1, bn[2]=2.0

(7) **Example**

(a) Solve the following simultaneous ordinary differential equations:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - y_1 \cdot y_2 \end{cases}$$

under the following boundary conditions:

$$y_1'|_{x=0.0} = 0.0, y_1|_{x=0.0} = 0.486, y_1'|_{x=3.0} = 2.0$$

(b) Input data

Name of function f(x, y, n, alf): f, n=2, xa=0.0, xb=3.0,
in[0]=0, ib[0]=1, ic[0]=1, bn[0]=0.0,
in[1]=0, ib[1]=1, ic[1]=0, bn[1]=0.486,
in[2]=1, ib[2]=1, ic[2]=1, bn[2]=2.0,
m[0]=2, m[1]=1, x, k=3, er, ea, nx, nev=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dosnnv */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n,double *alf)
#else
void    f(x,y,n,alf)
double *x;
double *y;
double *alf;
int    *n;
#endif
{
    y[2*(*n)]= y[1];
    y[(*n)+1]= y[(*n)]-(*alf)*(y[0]*y[1]);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double ax;
    double bx;
    int *in;
    int *ib;
    int *ic;
    double *bn;
    int *m;
    int n;
    double *x;
    int k;
    double epr;
    double epa;
    int nx;
    int nev;
    double *y;
    int isw;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,v,maxm,summ,nwk;
    FILE *fp;

    fp = fopen( "dosnnv.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dosnnv ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=2;
    k=3;
```

```
isw=0;
m  = ( int * )malloc((size_t)( sizeof(int) * n ));
if( m  == NULL )
{
    printf( "no enough memory for array m\n" );
    return -1;
}
printf( "\n\tOrder of Each Differential Equations\n\n" );
maxm=summ=0;
for( i=0 ; i<n ; i++ )
{
    fscanf( fp, "%d", &m[i] );
    printf( "\t  m[%6d] = %6d\n",i,m[i]);
    maxm=( (maxm>m[i]) ? maxm : m[i] );
    summ+=m[i];
}

in  = ( int * )malloc((size_t)( sizeof(int) * summ ));
if( in  == NULL )
{
    printf( "no enough memory for array in\n" );
    return -1;
}

ib  = ( int * )malloc((size_t)( sizeof(int) * summ ));
if( ib  == NULL )
{
    printf( "no enough memory for array ib\n" );
    return -1;
}

ic  = ( int * )malloc((size_t)( sizeof(int) * summ ));
if( ic  == NULL )
{
    printf( "no enough memory for array ic\n" );
    return -1;
}

bn  = ( double * )malloc((size_t)( sizeof(double) * summ ));
if( bn  == NULL )
{
    printf( "no enough memory for array  bn\n" );
    return -1;
}

x  = ( double * )malloc((size_t)( sizeof(double) * k ));
if( x  == NULL )
{
    printf( "no enough memory for array  x\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * (k*n*(maxm+1)) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * summ ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

fscanf( fp, "%lf", &ax );
fscanf( fp, "%lf", &bx );
printf( "\txa = %8.3g\n", ax );
printf( "\txb = %8.3g\n", bx );
printf( "\n\tBoundary Condition\n\n" );
for( i=0 ; i<summ ; i++ )
{
    fscanf( fp, "%d", &in[i] );
}
for( i=0 ; i<summ ; i++ )
{
    fscanf( fp, "%d", &ib[i] );
}
for( i=0 ; i<summ ; i++ )
{
    fscanf( fp, "%d", &ic[i] );
}
for( i=0 ; i<summ ; i++ )
{
    fscanf( fp, "%lf", &bn[i] );
}
printf( "\t  Position   Index of y   Order of y    Value\n" );
for( i=0 ; i<summ ; i++ )
{
    printf( "\t %6d     %6d       %6d      %8.3g\n",in[i],ib[i],ic[i],bn[i]);
}
printf( "\n\tSimulations Number of differential equations = %6d\n",n);
```

```
    printf( "\n\tPoints Where Approximate Values are Computed\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
        printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
    }
    printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",k );

    fscanf( fp, "%lf", &epr );
    fscanf( fp, "%lf", &epa );
    fscanf( fp, "%d", &nx );
    fscanf( fp, "%d", &nev );
    fscanf( fp, "%d", &isw );

    nwk=(n*maxm+1)*(n*maxm+1)*(nx+1)+n*maxm*( (3*n*maxm>maxm+15) ? 3*n*maxm : maxm+15 );
    wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
        return -1;
    }

    printf( "\tRequired Local Relative Precision = %8.3g\n", epr );
    printf( "\tRequired Local Absolute Precision = %8.3g\n", epa);
    printf( "\tMaximum Number of Shooting Points =   %6d\n", nx );
    printf( "\tMaximum Iterative Number = %6d\n", nev );
    printf( "\tisw = %6d\n", isw );

    fclose( fp );

    ierr = ASL_dosnnv(f, ax, bx, in, ib, ic, bn, m, n, x, k, epr, epa, &nx, &nev, y, isw, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tPractical Number of Shooting Points = %6d\n", nx );
    printf( "\tPractical Iterative Number = %6d\n", nev );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        for( j=0 ; j<n ; j++ )
        {
        for( v=0; v<m[j]+1 ; v++ )
        {
        printf( "\t  y[%6d][%6d][%6d] = %8.3g\n",i,j,v,y[i+k*j+k*n*v] );
        }
        }
    }

    free( in );
    free( ib );
    free( ic );
    free( bn );
    free( m );
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dosnnv ***

 ** Input **


Order of Each Differential Equations

  m[     0] =      2
  m[     1] =      1
xa =        0
xb =        3

Boundary Condition

  Position   Index of y   Order of y    Value
      0          1            1            1
      0          1            0        0.486
      1          1            1            2

Simulations Number of differential equations =      2

Points Where Approximate Values are Computed

  x[     0] =        0
  x[     1] =      1.5
  x[     2] =        3
```

```
Number of Points Where Approximate Values are Computed =      3
Required Local Relative Precision =        0
Required Local Absolute Precision =        0
Maximum Number of Shooting Points =       50
Maximum Iterative Number =      0
isw =      1

 ** Output **

ierr =      0

Practical Number of Shooting Points =     19
Practical Iterative Number =     19

Approximate Values

  y[     0][     0][     0] =    0.486
  y[     0][     0][     1] =        1
  y[     0][     0][     2] =   -0.528
  y[     0][     1][     0] =   -0.528
  y[     0][     1][     1] =     1.26
  y[     1][     0][     0] =     1.94
  y[     1][     0][     1] =     1.19
  y[     1][     0][     2] =    0.513
  y[     1][     1][     0] =    0.513
  y[     1][     1][     1] =    0.199
  y[     2][     0][     0] =     4.34
  y[     2][     0][     1] =        2
  y[     2][     0][     2] =    0.486
  y[     2][     1][     0] =    0.486
  y[     2][     1][     1] =   -0.111
```

2.3.2 ASL_dosnnf, ASL_rosnnf
High-Order Simultaneous Ordinary Differential Equations (Function Boundary Conditions)

(1) **Function**

ASL_dosnnf or ASL_rosnnf uses the multipoint shooting method to solve a boundary value problem for high-order simultaneous ordinary differential equations for which boundary conditions are given as functions.

(2) **Usage**

Double precision:

```
ierr = ASL_dosnnf  (f, fb, xa, xb, m, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rosnnf  (f, fb, xa, xb, m, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n, alf) that defines the differential equations as functions of x and y.
2	fb	–	–	Input	Name of function fb(ya, yb, n, g) that defines boundary conditions.
3	xa	{ D / R }	1	Input	x coordinate of left-hand side boundary
4	xb	{ D / R }	1	Input	x coordinate of right-hand side boundary
5	m	I*	n	Input	Differential order of the left-hand side of each of the simultaneous equations
6	n	I	1	Input	Number of simultaneous equations
7	x	{ D* / R* }	k	Input	x coordinate x_i where approximate solutions are calculated
8	k	I	1	Input	Number of calculation points
9	er	{ D / R }	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	ea	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local absolute precision (Default value: Unit for determining error)
11	nx	I*	1	Input	Maximum number of shooting points
				Output	Actual number of iterations
12	nev	I*	1	Input	Maximum number of iterations (Default value:100)
				Output	Actual number of iterations
13	y	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Output	Solutions $y_j^{(v)}(x_i)$ $\mathrm{y}[(\mathrm{i}-1)+\mathrm{k}\times(\mathrm{j}-1)+\mathrm{k}\times\mathrm{n}\times\mathrm{v}]$ $=y_j^{(v)}(x_i)$ **Size:** $\mathrm{k}\times\mathrm{n}\times(\max(\mathrm{m}[\mathrm{i}-1])+1)$
14	isw	I	1	Input	Parameterization processing switch 0: Parameterization not performed Nonzero: Parameterization performed
15	iwk	I*	See Contents	Work	Work area **Size**: $\sum_{\mathrm{i}=1}^{\mathrm{n}}\mathrm{m}[\mathrm{i}-1]$
16	wk	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**:$(\mathrm{n}\times\mathrm{km}+1)^2\times(\mathrm{nx}+1)+\mathrm{n}\times\mathrm{km}\times$ $\max(3\times\mathrm{n}\times\mathrm{km},\mathrm{km}+15)$ where $\mathrm{km}=\max(\mathrm{m}[\mathrm{i}-1])$
17	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{xa}<\mathrm{xb}$

(b) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(c) ea $\geq$ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) $\mathrm{nev}>0$ (Except when 0.0 is entered in order to set nev to the default value)

(e) $\mathrm{n}\geq 1$

(f) $\mathrm{m}[\mathrm{i}-1]\geq 1\ (\mathrm{i}=1,2,\cdots,\mathrm{n})$

(g) $\mathrm{k}\geq 1$

(h) $\mathrm{xa}\leq \mathrm{x}[\mathrm{i}-1]\leq \mathrm{xb}\ (\mathrm{i}=1,2,\cdots,\mathrm{k})$

(i) $\mathrm{nx}\geq\min(5\mathrm{i}+1,51)$
Where, i is minimum integer for which $\mathrm{xb}-\mathrm{xa}\leq \mathrm{i}$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where xa ≠ xb)	Processing is performed assuming xa and xb are replaced each other.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with the corresponding default value set.
3000	xa = xb	Processing is aborted.
3010	Restriction (e) was not satisfied.	
3020	Restriction (f) was not satisfied.	
3030	Restriction (g) was not satisfied.	
3040	Restriction (h) was not satisfied.	
3050	Restriction (i) was not satisfied.	
4000	The shooting point could not be added due to severe oscillation (large derivative).	
4500	The step size became too small during the initial value problem calculation (existence of singularity, etc).	
5000	The maximum number of shooting points was reached.	
5500	The maximum number of iterations was reached (including cases when solution does not exist or is indefinite).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) In a nonlinear problem for high-order simultaneous ordinary differential equations expressed as follows:

$$\begin{cases} y_1^{(m_1)} &= f_1(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \vdots & \vdots \\ y_i^{(m_i)} &= f_i(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \\ \vdots & \vdots \\ y_n^{(m_n)} &= f_n(x, y_1, \cdots, y_i, \cdots, y_i^{(j)}, \cdots, y_i^{(m_i-1)}, \cdots, y_n^{(m_n-1)}) \end{cases}$$

(Where, if left-hand side is $y_i^{(m_i)}$, then differential order j of corresponding right-hand side $y_i^{(j)}$ must satisfy $j \leq m_i - 1$.) if there is a multiplication or division of two or more of the variables $y_i^{(j)}$ in the terms on the right-hand side of any of these equations such as $y_1 \cdot y_1'$, $\frac{y_2}{y_3'}$, y_4^2 or a function other than a sum or scalar multiple of $y_i^{(j)}$ such as $\sin(y_1)$ or $\mathrm{abs}(y_2')$, parameterize this nonlinear problem by multiplying this portion by alf.

Example :

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - y_1 \cdot y_2 \end{cases}$$

Parameterize this nonlinear problem as follows:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

When the problem has been parameterized, set isw = 1. A linear problem need not be parameterized. In this case, set isw = 0.

(b) The actual name of function f(x, y, n, alf), that defines the differential equations, must be declared in the user program, and the actual function must be created.

The function f(x, y, n, alf) (in double-precision) for the high-order simultaneous ordinary differential equations that were parameterized as described in Note (a) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n, double *alf)
{
   y[(*n) * m[0]] = f1(*alf, x, y1, ..., yi, ..., yi^(j), ..., yi^(mi-1), ...);
      :
   y[i - 1 + (*n) * m[i - 1]] = fi(*alf, x, y1, ..., yi, ..., yi^(j), ..., yi^(mi-1), ...);
      :
   y[(*n) - 1 + (*n) * m[(*n) - 1]] = fn(*alf, x, y1, ..., yi, ..., yi^(j), ..., yi^(mi-1), ...);
}
```

where the following correspondences are assumed:
$x \leftrightarrow$ *x, $n \leftrightarrow$ *n, $y_i \leftrightarrow$ y[i − 1], $y_i^{(j)} \leftrightarrow$ y[i − 1 + (*n) * j]
$f_i(*\mathrm{alf}, x, \cdots)$ is parameterized one for $f_i(x, \cdots)$.

Example:

When parameterized ordinary differential equations are as follows:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

define the function as follows:

```
void FORTRAN f(double *x, double *y, int *n, double *alf)
{
   y[4] = y[1];
   y[3] = y[2] - (*alf) * (y[0] * y[1]);
}
```

The Input arguments: n=2, m[0]=2, m[1]=1, isw=1.

(c) The actual name of function fb(ya, yb, n, g), that defines the boundary conditions, must be declared in the user program, and the actual function must be created.

When q boundary conditions are given as follows using values $y_i^{(j)}(a)$ of $y_i^{(j)}$ at left-hand side boundary $x = a$ and values $y_i^{(j)}(b)$ of $y_i^{(j)}$ at right–hand side boundary $x = b$:

$$\begin{cases} g_1(y_1(a), \cdots, y_i^{(j)}(a), \cdots, y_n^{(m_n-1)}(a), y_1(b), \cdots, y_i^{(j)}(b), \cdots, y_n^{(m_n-1)}(b)) = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ g_v(y_1(a), \cdots, y_i^{(j)}(a), \cdots, y_n^{(m_n-1)}(a), y_1(b), \cdots, y_i^{(j)}(b), \cdots, y_n^{(m_n-1)}(b)) = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ g_q(y_1(a), \cdots, y_i^{(j)}(a), \cdots, y_n^{(m_n-1)}(a), y_1(b), \cdots, y_i^{(j)}(b), \cdots, y_n^{(m_n-1)}(b)) = 0 \end{cases}$$

the function fb(ya, yb, n, g) (in double-precision) should be created as follows:

```
void FORTRAN fb(double *ya, double *yb, int *n, double *g)
{
   g[0] = g1(y1(a), ..., yi^(j)(a), ..., yn^(mn-1)(a), y1(b), ..., yi^(j)(b), ..., yn^(mn-1)(b));
```

$$g[1] = g_2(y_1(a), \cdots, y_i^{(j)}(a), \cdots, y_n^{(m_n-1)}(a), y_1(b), \cdots, y_i^{(j)}(b), \cdots, y_n^{(m_n-1)}(b));$$
$$\vdots$$
$$g[q-1] = g_q(y_1(a), \cdots, y_i^{(j)}(a), \cdots, y_n^{(m_n-1)}(a), y_1(b), \cdots, y_i^{(j)}(b), \cdots, y_n^{(m_n-1)}(b));$$
}

where the following correspondences are assumed:
$n \leftrightarrow *n$, $y_i(a) \leftrightarrow ya[i-1]$, $y_i^{(j)}(a) \leftrightarrow ya[i-1+(*n)*j]$
$y_i(b) \leftrightarrow yb[i-1]$, $y_i^{(j)}(b) \leftrightarrow yb[i-1+(*n)*j]$

Example:

When boundary conditions are given as follows:

$$\begin{cases} y_1(a) - y_2(b) = 0.0 \\ y_1'(a) = 1.0 \\ y_1'(b) = 2.0 \end{cases}$$

define the function as follows:

```
void FORTRAN fb(double *ya, double *ya, int *n, double *g)
{
    g[0] = ya[0] - yb[1];
    g[1] = ya[2] - 1.0;
    g[2] = yb[2] - 2.0;
}
```

(7) **Example**

(a) Problem

Solve the following simultaneous ordinary differential equations:

$$\begin{cases} y_1'' = y_2 \\ y_2' = y_1' - y_1 \cdot y_2 \end{cases}$$

under the following boundary conditions:

$$\begin{cases} y_1(0.0) - y_2(3.0) = 0.0 \\ y_1'(0.0) = 1.0 \\ y_1'(3.0) = 2.0 \end{cases}$$

(b) Input data

Name of function f(x, y, n, alf): f1, name of function fb(ya, yb, n, g): f2, n=2, xa=0.0, xb=3.0, m[0]=2, m[1]=1, x, k=3, er, ea, nx, nev=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dosnnf */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f1(double *x,double *y,int *n,double *alf)
#else
void    f1(x,y,n,alf)
double *x;
double *y;
double *alf;
int    *n;
#endif
{
```

```
    y[2*(*n)]= y[1];
    y[(*n)+1]= y[(*n)]-(*alf)*(y[0]*y[1]);
}
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f2(double *ya,double *yb,int *n,double *g)
#else
void    f2(ya,yb,n,g)
double *ya;
double *yb;
double *g;
int    *n;
#endif
{
    g[0]= ya[0]-yb[1];
    g[1]= ya[(*n)]-1.0;
    g[2]= yb[(*n)]-2.0;
}
#ifdef __cplusplus
}
#endif

int main()
{
    double ax;
    double bx;
    int *m;
    int n;
    double *x;
    int k;
    double epr;
    double epa;
    int nx;
    int nev;
    double *y;
    int isw;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,v,maxm,summ,nwk;
    FILE *fp;

    fp = fopen( "dosnnf.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dosnnf ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=2;
    k=3;

    m  = ( int * )malloc((size_t)( sizeof(int) * n ));
    if( m  == NULL )
    {
        printf( "no enough memory for array m\n" );
        return -1;
    }
    x  = ( double * )malloc((size_t)( sizeof(double) * k ));
    if( x  == NULL )
    {
        printf( "no enough memory for array  x\n" );
        return -1;
    }


    fscanf( fp, "%lf", &ax );
    fscanf( fp, "%lf", &bx );
    printf( "\txa  = %8.3g\n", ax );
    printf( "\txb  = %8.3g\n", bx );
    printf( "\n\tBoundary Conditions\n\n" );
    printf( "\t  ya1-yb2 = 0.0 \n" );
    printf( "\t  ya1'    = 1.0    (ya=y(x=0.0),yb=y(x=3.0))\n" );
    printf( "\t  yb1'    = 2.0 \n" );
    printf( "\n\tOrder of Each Differential Equations\n\n" );
    maxm=summ=0;
    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%d", &m[i] );
        printf( "\t  m[%6d]=%6d\n",i,m[i]);
        maxm=( (maxm>m[i]) ? maxm : m[i] );
```

```
        summ+=m[i];
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * (k*n*(maxm+1)) ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y \n" );
        return -1;
    }
    iwk  = ( int * )malloc((size_t)( sizeof(int) * summ ));
    if( iwk  == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    printf( "\n\tSimulations Number of differential equations = %6d\n",n);
    printf( "\n\tPoints Where Approximate Values are Computed\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
        printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
    }
    printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",k );

    fscanf( fp, "%lf", &epr );
    fscanf( fp, "%lf", &epa );
    fscanf( fp, "%d", &nx );
    fscanf( fp, "%d", &nev );
    fscanf( fp, "%d", &isw );

    printf( "\n\tRequired Local Relative Precision  = %8.3g\n", epr );
    printf( "\tRequired Local Absolute Precision  = %8.3g\n", epa);
    printf( "\tMaximum Number of Shooting Points  =   %6d\n", nx );
    printf( "\tMaximum Iterative Number  = %6d\n", nev );
    printf( "\tisw = %6d\n", isw );

    nwk=(n*maxm+1)*(n*maxm+1)*(nx+1)+n*maxm*( (3*n*maxm>maxm+15) ? 3*n*maxm : maxm+15 );
    wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
        return -1;
    }
    fclose( fp );

    ierr = ASL_dosnnf(f1, f2, ax, bx, m, n, x, k, epr, epa, &nx, &nev, y, isw, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tPractical Number of Shooting Points = %6d\n", nx );
    printf( "\n\tPractical Iterative Number = %6d\n", nev );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<3 ; i++ )
    {
        for( j=0 ; j<n ; j++ )
        {
        for( v=0; v<m[j]+1 ; v++ )
        {
        printf( "\t  y[%6d][%6d][%6d] = %8.3g\n",i,j,v,y[i+k*j+k*n*v] );
        }
        }
    }

    free( m );
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dosnnf ***

 ** Input **

xa  =        0
xb  =        3

Boundary Conditions

  ya1-yb2 = 0.0
  ya1'    = 1.0    (ya=y(x=0.0),yb=y(x=3.0))
  yb1'    = 2.0

Order of Each Differential Equations
```

```
  m[     0]=     2
  m[     1]=     1

Simulations Number of differential equations =      2

Points Where Approximate Values are Computed

  x[     0] =        0
  x[     1] =      1.5
  x[     2] =        3

Number of Points Where Approximate Values are Computed =      3

Required Local Relative Precision  =        0
Required Local Absolute Precision  =        0
Maximum Number of Shooting Points  =       50
Maximum Iterative Number  =      0
isw =      1

 ** Output **

ierr =      0

Practical Number of Shooting Points =     19

Practical Iterative Number =     19

Approximate Values

  y[     0][     0][     0] =    0.486
  y[     0][     0][     1] =        1
  y[     0][     0][     2] =   -0.528
  y[     0][     1][     0] =   -0.528
  y[     0][     1][     1] =     1.26
  y[     1][     0][     0] =     1.94
  y[     1][     0][     1] =     1.19
  y[     1][     0][     2] =    0.513
  y[     1][     1][     0] =    0.513
  y[     1][     1][     1] =    0.199
  y[     2][     0][     0] =     4.34
  y[     2][     0][     1] =        2
  y[     2][     0][     2] =    0.486
  y[     2][     1][     0] =    0.486
  y[     2][     1][     1] =   -0.111
```

2.3.3 ASL_dofnnv, ASL_rofnnv
First-Order Simultaneous Ordinary Differential Equations (Numerical Boundary Conditions)

(1) **Function**

ASL_dofnnv or ASL_rofnnv uses the multipoint shooting method to solve a boundary value problem for first-order simultaneous ordinary differential equations for which boundary conditions are given as input values.

(2) **Usage**

Double precision:

```
ierr = ASL_dofnnv  (f, xa, xb, in, ib, bn, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rofnnv  (f, xa, xb, in, ib, bn, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n, alf) that defines the differential equations as a function of x and y.
2	xa	{ D / R }	1	Input	x coordinate of left-hand side boundary
3	xb	{ D / R }	1	Input	x coordinate of right-hand side boundary
4	in	I*	n	Input	Positions where boundary conditions are set (xa side:0, xb side: Nonzero)
5	ib	I*	n	Input	Element numbers i of y_i which gives boundary conditions value
6	bn	{ D* / R* }	n	Input	Boundary condition values given for variable y_i.

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
7	n	I	1	Input	Number of simultaneous equations
8	x	{D∗ R∗}	k	Input	x coordinate x_i where approximate solutions are calculated
9	k	I	1	Input	Number of calculation points
10	er	{D R}	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
11	ea	{D R}	1	Input	Required local absolute precision (Default value: Unit for determining error)
12	nx	I*	1	Input	Maximum number of shooting points
				Output	Actual number of shooting points
13	nev	I*	1	Input	Maximum number of iterations (Default value: 100)
				Output	Actual number of iterations
14	y	{D∗ R∗}	See Contents	Output	Solutions $y_j^{(v)}(x_i)$ y[(i − 1) + k × (j − 1) + k × n × v] $= y_j^{(v)}(x_i)$ **Size:** k×n×2
15	isw	I	1	Input	Parameterization processing switch 0: Parameterization not performed Nonzero: Parameterization performed
16	iwk	I*	n	Work	Work area
17	wk	{D∗ R∗}	See Contents	Work	Work area **Size**: $(n+1)^2 \times (nx+1) + n \times \max(2 \times n, 17)$
18	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa < xb

(b) er≥ e_r. where e_r= double precision:10^{-14}, single precision: 10^{-5}
(Except when 0.0 is entered in order to set er to the default value)

(c) ea ≥ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) nev > 0 (Except when 0 is entered in order to set nev to the default value)

(e) n ≥ 1

(f) $1 \le ib[i-1] \le n$ $(i = 1, 2, \cdots, \sum_{j=1}^{n} m[j-1])$

(g) k ≥ 1

(h) $xa \le x[i-1] \le xb$ $(i = 1, 2, \cdots, k)$

(i) $nx \ge \min(5i + 1, 51)$
where i is the minimum integer for which $xb - xa \le i$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where xa $\neq$ xb)	Processing is performed assuming xa and xb are replaced each other.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is aborted.
3000	xa=xb	
3010	Restriction (e) was not satisfied.	
3020	Restriction (f) was not satisfied.	
3030	Restriction (g) was not satisfied.	
3040	Restriction (h) was not satisfied.	
3050	Restriction (i) was not satisfied.	
4000	The shooting point could not be added due to severe oscillation (large derivative), etc.	Processing is aborted.
4500	The step size became too small during the initial value problem calculation (existence of singularity, etc).	
5000	The maximum number of shooting points was reached.	
5500	The maximum number of iterations was reached (including cases when solution does not exist or is indefinite).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) In a nonlinear problem for first-order simultaneous ordinary differential equations expressed as follows:

$$\begin{cases} y_1' &= f_1(x, y_1, y_2, \cdots, y_n) \\ y_2' &= f_2(x, y_1, y_2, \cdots, y_n) \\ \vdots & \vdots \qquad\qquad \vdots \\ y_n' &= f_n(x, y_1, y_2, \cdots, y_n) \end{cases}$$

if there is a multiplication or division of two or more of the variables y_i in the terms on the right-hand side of any of these equations such as $y_1 \cdot y_2$, $\frac{y_2}{y_3}$ or y_4^2 or a function other than a sum or scalar multiple of y_i such as $\sin(y_1)$ or $\mathrm{abs}(y_2)$, parameterize this nonlinear problem by multiplying this portion by alf.
Example :

$$\begin{cases} y_1' = y_2 \\ y_2' = y_1 - y_1 \cdot y_2 \end{cases}$$

Parameterize this nonlinear problem as follows:

$$\begin{cases} y_1' = y_2 \\ y_2' = y_1 - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

When the problem has been parameterized, set isw = 1.
A linear problem need not be parameterized. In this case, set isw = 0.

(b) The actual name of function f(x, y, n, alf), that defines the differential equations, must be declared in the user program, and the actual function must be created.

The function f(x, y, n, alf) (in double-precision) for the first-order simultaneous ordinary differential equations that were parameterized as described in Note (a) should be created as follows:

void FORTRAN f(double *x, double *y, int *n, double *alf)
{
$y[(*n)] = f_1(*alf, x, y_1, \cdots, y_i, \cdots, y_n);$
$\vdots$
$y[i-1+(*n)] = f_i(*alf, x, y_1, \cdots, y_i, \cdots, y_n);$
$\vdots$
$y[2*(*n)-1] = f_n(*alf, x, y_1, \cdots, y_i, \cdots, y_n);$
}

where the following correspondences are assumed:
$x \leftrightarrow *x$, $n \leftrightarrow *n$, $y_i \leftrightarrow y[i-1]$
$f_i(*alf, x, \cdots)$ is parameterized one for $f_i(x, \cdots)$.
Example:
When parameterized ordinary differential equations are as follows:

$$\begin{cases} y_1' = y_2 \\ y_2' = y_1 - \underline{alf \cdot y_1 \cdot y_2} \end{cases}$$

define the function as follows:

void FORTRAN f(double *x, double *y, int *n, double *alf)
{
y[2] = y[1];
y[3] = y[0] − (*alf) * (y[0] * y[1]);
}

The Input arguments: n=2, isw=1.

(c) If the boundary conditions of the first-order simultaneous ordinary differential equations are given by:

$$\text{left-hand side boundary} \begin{cases} y_{a_1} = c_1 \\ y_{a_2} = c_2 \\ \vdots \\ y_{a_p} = c_p \end{cases} \quad \text{right-hand side boundary} \begin{cases} y_{a_{p+1}} = c_{p+1} \\ y_{a_{p+2}} = c_{p+2} \\ \vdots \\ y_{a_n} = c_n \end{cases}$$

set array in, ib and bn as follows: $in[i-1] = 0$ $(i = 1, 2, \cdots, p)$
$in[i-1] = 1$ $(i = p+1, p+2, \cdots, n)$
$ib[i-1] = a_i (i = 1, 2, \cdots, n)$
$bn[i-1] = c_i (i = 1, 2, \cdots, n)$
Example: If the boundary conditions are given by:

left-hand side boundary $y_1 = 1.0$ right-hand side boundary $y_2 = 2.0$

the input values of in, ib and bn are as follows:
in[0]=0, ib[0]=1, bn[0]=1.0
in[1]=1, ib[1]=2, bn[1]=2.0

(7) **Example**

(a) Problem

Solve the following simultaneous ordinary differential equations:

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1' - y_1 \cdot y_2 \end{cases}$$

under the following boundary conditions:

$$y_2|_{x=0.0} = 1.0, y_2|_{x=1.0} = 2.0$$

(b) Input data

Name of function f(x, y, n, alf): f, n=2, xa=0.0, xb=1.0,
in[0]=0, ib[0]=2, bn[0]=1.0,
in[1]=1, ib[1]=2, bn[1]=2.0,
x, k=3, er, ea, nx, nev=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dofnnv */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *n,double *alf)
#else
void    f(x,y,n,alf)
double *x;
double *y;
double *alf;
int    *n;
#endif
{
    y[(*n)]  = y[1];
    y[(*n)+1]= -y[0]-(*alf)*(y[0]*y[1]);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double ax;
    double bx;
    int *in;
    int *ib;
    double *bn;
    int mx;
    double *x;
    int m;
    double epr;
    double epa;
    int nx;
    int nev;
    double *y;
    int isw;
    int *iwk;
    double *wk;
    int ierr;
    int nwk;
    int i,j,k;
    FILE *fp;

    fp = fopen( "dofnnv.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dofnnv ***\n" );
    printf( "\n     ** Input **\n\n" );
    mx=2;
    m=3;
```

```
in  = ( int * )malloc((size_t)( sizeof(int) * mx ));
if( in  == NULL )
{
    printf( "no enough memory for array in\n" );
    return -1;
}

ib  = ( int * )malloc((size_t)( sizeof(int) * mx ));
if( ib  == NULL )
{
    printf( "no enough memory for array ib\n" );
    return -1;
}

bn  = ( double * )malloc((size_t)( sizeof(double) * mx ));
if( bn  == NULL )
{
    printf( "no enough memory for array  bn\n" );
    return -1;
}
x  = ( double * )malloc((size_t)( sizeof(double) * m ));
if( x  == NULL )
{
    printf( "no enough memory for array  x\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * (m*mx*2) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

fscanf( fp, "%lf", &ax );
fscanf( fp, "%lf", &bx );
printf( "\txa  = %8.3g\n", ax );
printf( "\txb  = %8.3g\n", bx );
printf( "\n\tBoundary Condition\n\n" );
for( i=0 ; i<mx ; i++ )
{
    fscanf( fp, "%d", &in[i] );
}
for( i=0 ; i<mx ; i++ )
{
    fscanf( fp, "%d", &ib[i] );
}
for( i=0 ; i<mx ; i++ )
{
    fscanf( fp, "%lf", &bn[i] );
}
printf( "\t  Position   Index of y       Value\n" );
for( i=0 ; i<mx ; i++ )
{
    printf( "\t%6d     %6d         %8.3g\n",in[i],ib[i],bn[i]);
}
printf( "\n\tSimulations Number of Differential Equations = %6d\n",mx);
printf( "\n\tPoints Where Approximate Values are Computed \n" );
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
    printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
}
printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",m );

fscanf( fp, "%lf", &epr );
fscanf( fp, "%lf", &epa );
fscanf( fp, "%d", &nx );
fscanf( fp, "%d", &nev );
fscanf( fp, "%d", &isw );

nwk = (mx+1)*(mx+1)*(nx+1)+mx*( (2*mx> 17) ? 2*mx :  17 );
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * mx ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

printf( "\n\tRequired Local Relative Precision  = %8.3g\n", epr );
printf( "\tRequired Local Absolute Precision  = %8.3g\n", epa);
printf( "\tMaximum Number of Shooting Points  =   %6d\n", nx );
printf( "\tMaximum Iterative Number  = %6d\n", nev );
printf( "\tisw =  %6d\n", isw );
```

```
    fclose( fp );

    ierr = ASL_dofnnv(f, ax, bx, in, ib, bn, mx, x, m, epr, epa, &nx, &nev, y, isw, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tPractical Number of Shooting Points = %6d\n", nx );
    printf( "\tPractical Iterative Number = %6d\n", nev );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<m ; i++ )
    {
        for( j=0 ; j<mx ; j++ )
        {
        for( k=0; k<2 ; k++ )
        {
        printf( "\t  y[%6d][%6d][%6d] = %8.3g\n",i,j,k,y[i+m*(j+mx*k)] );
        }
        }
    }

    free( in );
    free( ib );
    free( bn );
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dofnnv ***

 ** Input **

xa  =        0
xb  =        1

Boundary Condition

  Position   Index of y       Value
     0          2               1
     1          2               2

Simulations Number of Differential Equations =      2

Points Where Approximate Values are Computed
  x[     0] =        0
  x[     1] =      0.5
  x[     2] =        1

Number of Points Where Approximate Values are Computed =      3

Required Local Relative Precision  =        0
Required Local Absolute Precision  =        0
Maximum Number of Shooting Points  =       50
Maximum Iterative Number  =      0
isw =       1

 ** Output **

ierr =      0

Practical Number of Shooting Points =      6
Practical Iterative Number =     20

Approximate Values

  y[     0][     0][     0] =    -1.27
  y[     0][     0][     1] =        1
  y[     0][     1][     0] =        1
  y[     0][     1][     1] =     2.54
  y[     1][     0][     0] =    -0.46
  y[     1][     0][     1] =     2.16
  y[     1][     1][     0] =     2.16
  y[     1][     1][     1] =     1.45
  y[     2][     0][     0] =    0.655
  y[     2][     0][     1] =        2
  y[     2][     1][     0] =        2
  y[     2][     1][     1] =    -1.96
```

2.3.4 ASL_dofnnf, ASL_rofnnf
First-Order Simultaneous Ordinary Differential Equations (Function Boundary Conditions)

(1) **Function**

ASL_dofnnf or ASL_rofnnf uses the multipoint shooting method to solve a boundary value problem for first-order simultaneous ordinary differential equations for which boundary conditions are given as functions.

(2) **Usage**

Double precision:

```
ierr = ASL_dofnnf  (f, fb, xa, xb, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rofnnf  (f, fb, xa, xb, n, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, n, alf) that defines the differential equations as a function of x and y.
2	fb	–	–	Input	Name of function fb(ya, yb, n, g) that defines boundary conditions.
3	xa	{ D / R }	1	Input	x coordinate of left-hand side boundary
4	xb	{ D / R }	1	Input	x coordinate of right-hand side boundary
5	n	I	1	Input	Number of simultaneous equations
6	x	{ D∗ / R∗ }	k	Input	x coordinate x_i where approximate solutions are calculated
7	k	I	1	Input	Number of calculation points
8	er	{ D / R }	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
9	ea	{ D / R }	1	Input	Required local absolute precision (Default value: Unit for determining error)

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	nx	I*	1	Input	Maximum number of shooting points
				Output	Actual number of shooting points
11	nev	I*	1	Input	Maximum number of iterations (Default value: 100)
				Output	Actual number of iterations
12	y	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Output	Solutions $y_j^{(v)}(x_i)$ $\text{y}[(\text{i}-1)+\text{k}\times(\text{j}-1)+\text{k}\times\text{n}\times\text{v}]$ $= y_j^{(v)}(x_i)$ **Size:** k×n×2
13	isw	I	1	Input	Parametrization processing switch 0: Parametrization not performed Nonzero: Parametrization performed
14	iwk	I*	n	Work	Work area
15	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $(\text{n}+1)^2\times(\text{nx}+1)+\text{n}\times\max(2\times\text{n}, 17)$
16	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa < xb

(b) er≥ e_r. where e_r= double precision:10^{-14}, single precision: 10^{-5}
(Except when 0.0 is entered in order to set er to the default value)

(c) ea ≥ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) nev > 0 (Except when 0 is entered in order to set nev to the default value)

(e) n ≥ 1

(f) k ≥ 1

(g) $\text{xa} \le \text{x}[\text{i}-1] \le \text{xb}$ $(\text{i} = 1, 2, \cdots, \text{k})$

(h) $\text{nx} \ge \min(5\text{i}+1, 51)$
where i is the minimum integer for which $\text{xb}-\text{xa} \le \text{i}$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where xa ≠ xb.)	Processing is performed assuming xa and xb are replaced each other.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is aborted.

ierr value	Meaning	Processing
3000	xa=xb	Processing is aborted.
3010	Restriction (e) was not satisfied.	
3020	Restriction (f) was not satisfied.	
3030	Restriction (g) was not satisfied.	
3040	Restriction (h) was not satisfied.	
4000	The shooting point could not be added due to severe oscillation (large derivative), etc.	
4500	The step size became too small during the initial value problem calculation (existence of singularity, etc).	
5000	The maximum number of shooting points was reached.	
5500	The maximum number of iterations was reached (including cases when solution does not exist or is indefinite).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) In a nonlinear problem for first-order simultaneous ordinary differential equations expressed as follows:

$$\begin{cases} y_1' &= f_1(x, y_1, y_2, \cdots, y_n) \\ y_2' &= f_2(x, y_1, y_2, \cdots, y_n) \\ \vdots & \vdots \qquad\qquad \vdots \\ y_n' &= f_n(x, y_1, y_2, \cdots, y_n) \end{cases}$$

if there is a multiplication or division of two or more of the variables y_i in the terms on the right-hand side of any of these equations such as $y_1 \cdot y_2$, $\frac{y_2}{y_3}$ or y_4^2 or a function other than a sum or scalar multiple of y_i such as $\sin(y_1)$ or $\mathrm{abs}(y_2)$, parameterize this nonlinear problem by multiplying this portion by alf.

Example :

$$\begin{cases} y_1' = y_2 \\ y_2' = y_1 - y_1 \cdot y_2 \end{cases}$$

Parameterize this nonlinear problem as follows:

$$\begin{cases} y_1' = y_2 \\ y_2' = y_1 - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

When the problem has been parameterized, set isw = 1. A linear problem need not be parameterized. In this case, set isw = 0.

(b) The actual name of function f(x, y, n, alf), that defines the differential equations, must be declared in the user program, and the actual function must be created.

The function f(x, y, n, alf) (in double-precision) for the high-order simultaneous ordinary differential equations that were parameterized as described in Note (a) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *n, double *alf)
{
    y[(*n)] = f1(*alf, x, y1, ···, yi, ···, yn);
      ⋮
```

$$\begin{aligned}
&\mathrm{y[i-1+(*n)]} = f_i(*\mathrm{alf}, x, y_1, \cdots, y_i, \cdots, y_n);\\
&\quad\vdots\\
&\mathrm{y[2*(*n)-1]} = f_n(*\mathrm{alf}, x, y_1, \cdots, y_i, \cdots, y_n);
\end{aligned}$$

}

where the following correspondences are assumed:

$x \leftrightarrow *\mathrm{x}$, $n \leftrightarrow *\mathrm{n}$, $y_i \leftrightarrow \mathrm{y[i-1]}$

$f_i(*\mathrm{alf}, x, \cdots)$ is parameterized one for $f_i(x, \cdots)$.

Example:

When parameterized ordinary differential equations are as follows:

$$\begin{cases} y_1' = y_2 \\ y_2' = y_1 - \underline{\mathrm{alf} \cdot y_1 \cdot y_2} \end{cases}$$

define the function as follows:

```
void FORTRAN f(double *x, double *y, int *n, double *alf)
{
    y[2] = y[1];
    y[3] = y[0] - (*alf) * (y[0] * y[1]);
}
```

The Input arguments: n=2, isw=1.

(c) The actual name of function fb(ya, yb, n, g), that defines the boundary conditions, must be declared in the user program, and the actual function must be created.

When n boundary conditions are given as follows using values $y_i(a)$ of y_i at left-hand side boundary $x = a$ and values $y_i(b)$ of y_i at right–hand side boundary $x = b$:

$$\begin{cases} g_1(y_1(a), \cdots, y_i, \cdots, y_n(a), y_1(b), \cdots, y_i(b), \cdots, y_n(b)) = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ g_v(y_1(a), \cdots, y_i, \cdots, y_n(a), y_1(b), \cdots, y_i(b), \cdots, y_n(b)) = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ g_n(y_1(a), \cdots, y_i, \cdots, y_n(a), y_1(b), \cdots, y_i(b), \cdots, y_n(b)) = 0 \end{cases}$$

the function fb(ya, yb, n, g) (in double-precision) should be created as follows:

```
void FORTRAN fb(double *ya, double *yb, int *n, double *g)
{
    g[0] = g1(y1(a), ..., yi, ..., yn(a), y1(b), ..., yi(b), ..., yn(b));
    g[1] = g2(y1(a), ..., yi, ..., yn(a), y1(b), ..., yi(b), ..., yn(b));
      :
    g[n-1] = gn(y1(a), ..., yi, ..., yn(a), y1(b), ..., yi(b), ..., yn(b));
}
```

where the following correspondences are assumed.

$n \leftrightarrow *\mathrm{n}$, $y_i(a) \leftrightarrow \mathrm{ya[i-1]}$, $y_i(b) \leftrightarrow \mathrm{yb[i-1]}$

Example:

When boundary conditions are given as follows:

$$\begin{cases} y_1(a) - y_2(b) = 0.0 \\ y_2(a) = 1.0 \end{cases}$$

define the function as follows:

```
void FORTRAN fb(double *ya, double *yb, int *n, double *g)
{
    g[0] = ya[0] - yb[1];
    g[1] = ya[1] - 1.0;
}
```

(7) **Example**

(a) Problem

Solve the following simultaneous ordinary differential equations:

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 - y_1 \cdot y_2 \end{cases}$$

under the following boundary conditions:

$$\begin{cases} y_1(0.0) - y_2(1.0) = 0.0 \\ y_2(0.0) = 1.0 \end{cases}$$

(b) Input data

Name of function f(x, y, n, alf): f1 and name of function fb(ya, yb, n, g): f2, n=2, xa=0.0, xb=1.0, x, k=3, er, ea, nx, nev=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dofnnf */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f1(double *x,double *y,int *n,double *alf)
#else
void    f1(x,y,n,alf)
double *x;
double *y;
double *alf;
int    *n;
#endif
{
    y[(*n)]= y[1];
    y[(*n)+1]= -y[0]-(*alf)*(y[0]*y[1]);

}
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f2(double *ya,double *yb,int *n,double *g)
#else
void    f2(ya,yb,n,g)
double *ya;
double *yb;
double *g;
int    *n;
#endif
{
    g[0]= ya[0]-yb[1];
    g[1]= ya[1]-1.0;
}
#ifdef __cplusplus
}
#endif


int main()
{
    double ax;
```

```
double bx;
int n;
double *x;
int k;
double epr;
double epa;
int nx;
int nev;
double *y;
int isw;
int *iwk;
double *wk;
int ierr;
int i,j,v,nwk;
FILE *fp;

fp = fopen( "dofnnf.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dofnnf ***\n" );
printf( "\n     ** Input **\n\n" );
n=2;
k=3;

x  = ( double * )malloc((size_t)( sizeof(double) * k ));
if( x  == NULL )
{
    printf( "no enough memory for array  x\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * (k*n*2) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * n ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

fscanf( fp, "%lf", &ax );
fscanf( fp, "%lf", &bx );
printf( "\txa  =%8.3g\n", ax );
printf( "\txb  =%8.3g\n", bx );
printf( "\n\tBoundary Conditions\n" );
printf( "\t  ya1-yb2 = 0.0 \n" );
printf( "\t  ya2     = 1.0    (ya=y(x=0.0),yb=y(x=1.0))\n" );
printf( "\n\tSimulations Number of differential equations =%6d\n",n);
printf( "\n\tPoints Where Approximate Values are Computed \n" );
for( i=0 ; i<k ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
    printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
}
printf( "\n\tNumber of Points Where Approximate Values are Computed =%6d\n",k );

fscanf( fp, "%lf", &epr );
fscanf( fp, "%lf", &epa );
fscanf( fp, "%d", &nx );
fscanf( fp, "%d", &nev );
fscanf( fp, "%d", &isw );

nwk=(n+1)*(n+1)*(nx+1)+n*( (2*n>17) ? 2*n : 17 );
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}

printf( "\n\tRequired Local Relative Precision = %8.3g\n", epr );
printf( "\tRequired Local Absolute Precision = %8.3g\n", epa);
printf( "\tMaximum Number of Shooting Points =   %6d\n", nx );
printf( "\tMaximum Iterative Number = %6d\n", nev );
printf( "\tisw = %6d\n", isw );

fclose( fp );

ierr = ASL_dofnnf(f1, f2, ax, bx, n, x, k, epr, epa, &nx, &nev, y, isw, iwk, wk);

printf( "\n     ** Output **\n\n" );
```

```
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tPractical Number of Shooting Points = %6d\n", nx );
    printf( "\tPractical Iterative Number = %6d\n", nev );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        for( j=0 ; j<n ; j++ )
        {
        for( v=0; v<2 ; v++ )
        {
            printf( "\t  y[%6d][%6d][%6d] = %8.3g\n",i,j,v,y[i+k*j+k*n*v] );
        }
        }
    }

    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dofnnf ***

 ** Input **

xa  =       0
xb  =       1

Boundary Conditions
  ya1-yb2 = 0.0
  ya2     = 1.0    (ya=y(x=0.0),yb=y(x=1.0))

Simulations Number of differential equations =     2

Points Where Approximate Values are Computed
  x[     0] =        0
  x[     1] =      0.5
  x[     2] =        1

Number of Points Where Approximate Values are Computed =     3

Required Local Relative Precision =        0
Required Local Absolute Precision =        0
Maximum Number of Shooting Points =       50
Maximum Iterative Number =      0
isw =      1

 ** Output **

ierr =      0

Practical Number of Shooting Points =      6
Practical Iterative Number =     18

Approximate Values

  y[     0][     0][     0] =    0.159
  y[     0][     0][     1] =        1
  y[     0][     1][     0] =        1
  y[     0][     1][     1] =   -0.317
  y[     1][     0][     0] =    0.584
  y[     1][     0][     1] =    0.649
  y[     1][     1][     0] =    0.649
  y[     1][     1][     1] =   -0.963
  y[     2][     0][     0] =    0.785
  y[     2][     0][     1] =    0.159
  y[     2][     1][     0] =    0.159
  y[     2][     1][     1] =   -0.909
```

2.3.5 ASL_dohnnv, ASL_rohnnv
High-Order Ordinary Differential Equation (Numerical Boundary Conditions)

(1) **Function**

ASL_dohnnv or ASL_rohnnv uses the multipoint shooting method to solve a boundary value problem for a high-order ordinary differential equation for which boundary conditions are given as input values.

(2) **Usage**

Double precision:

```
ierr = ASL_dohnnv  (f, xa, xb, in, ic, bn, m, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rohnnv  (f, xa, xb, in, ic, bn, m, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, m, alf) that defines the differential equations as a function of x and y.
2	xa	{D / R}	1	Input	x coordinate of left-hand side boundary
3	xb	{D / R}	1	Input	x coordinate of right-hand side boundary
4	in	I*	m	Input	Positions where boundary conditions are set (xaside :0, xbside: Nonzero)
5	ic	I*	m	Input	Differential order j of variable $y^{(j)}$ which gives boundary conditions value
6	bn	{D*/ R*}	m	Input	Boundary condition values given for variable $y^{(j)}$
7	m	I	1	Input	Maximum differential order of variable y in differential equations
8	x	{D*/ R*}	k	Input	x coordinate x_i where approximate solutions are calculated
9	k	I	1	Input	Number of calculation points

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	er	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
11	ea	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Required local absolute precision (Default value: Unit for determining error)
12	nx	I*	1	Input	Maximum number of shooting points
				Output	Actual number of shooting points
13	nev	I*	1	Input	Maximum number of iterations (Default value:100)
				Output	Actual number of iterations
14	y	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	$\mathrm{k}\times(\mathrm{m}+1)$	Output	Solutions $y^{(v)}(x_i)$ $\mathrm{y}[(\mathrm{i}-1)+\mathrm{k}\times\mathrm{v}] = y^{(v)}(x_i)$
15	isw	I	1	Input	Parameterization processing switch 0: Parameterization not performed Nonzero: Parameterization performed
16	iwk	I*	m	Work	Work area
17	wk	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $(\mathrm{m}+1)^2\times(\mathrm{nx}+1)+\mathrm{m}\times\max(2\times\mathrm{m}+1, 16)$
18	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{xa} < \mathrm{xb}$

(b) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(c) ea $\geq$ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) $\mathrm{nev} > 0$ (Except when 0 is entered in order to set nev to the default value)

(e) $\mathrm{m} \geq 1$

(f) $0 \leq \mathrm{ic}[\mathrm{i}-1] < \mathrm{m}$ $(\mathrm{i} = 1, 2, \cdots, \mathrm{m})$

(g) $\mathrm{k} \geq 1$

(h) $\mathrm{xa} \leq \mathrm{x}[\mathrm{i}-1] \leq \mathrm{xb}$ $(\mathrm{i} = 1, 2, \cdots, \mathrm{k})$

(i) $\mathrm{nx} \geq \min(5\mathrm{i}+1, 51)$
Where, i is minimum integer for which $\mathrm{xb}-\mathrm{xa} \leq \mathrm{i}$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where xa $\neq$ xb)	Processing is performed assuming xa and xb are replaced each other.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with the corresponding default value set.
3000	xa=xb	Processing is aborted.
3010	Restriction (e) was not satisfied.	
3020	Restriction (f) was not satisfied.	
3030	Restriction (g) was not satisfied.	
3040	Restriction (h) was not satisfied.	
3050	Restriction (i) was not satisfied.	
4000	The shooting point could not be added due to severe oscillation (large derivative), etc.	
4500	The step size became too small during the initial value problem calculation (existence of singularity, etc.)	
5000	The maximum number of shooting points was reached.	
5500	The maximum number of iterations was reached (including cases when solution does not exist or is indefinite).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) In a nonlinear problem for a high-order ordinary differential equation expressed as follows:

$$y^{(m)} = f(x, y, y', \cdots, y^{(j)}, \cdots, y^{(m-1)})$$

(Where, differential order j of right-hand side $y^{(j)}$ must satisfy $j \leq m-1$.) if there is a multiplication or division of two or more of the variables $y^{(j)}$ in the terms on the right-hand side of the equation such as $y \cdot y'$, $\frac{y}{y'}$, y^2, or a function other than a sum or scalar multiple of $y^{(j)}$ such as sin(y) or abs(y'), parameterize this nonlinear problem by multiplying this portion by alf.

Example :

$$y'' = y' + y^2$$

Parameterize this nonlinear problem as follows:

$$y'' = y' + \underline{\text{alf} \cdot y^2}$$

When the problem has been parameterized, set isw = 1.
A linear problem need not be parameterized. In this case, set isw = 0.

(b) The actual name of function f(x, y, m, alf), that defines the differential equations, must be declared in the user program, and the actual function must be created.

The function f(x, y, m, alf) (in double-precision) for the high-order ordinary differential equation that was parameterized as described in Note (a) should be created as follows:

void FORTRAN f(double *x, double *y, int *m, double *alf)
{
y[*m − 1] = f(*alf, $x, y, \cdots, y^{(j)}, \cdots, y^{(m-1)}$)
}

where the following correspondences are assumed.
$x \leftrightarrow$ *x, $m \leftrightarrow$ *m, $y \leftrightarrow$ y[0], $y^{(j)} \leftrightarrow$ y[j]
f(*alf, $x, \cdots$) is parameterized one for $f(x, \cdots)$.
Example:
When parameterized ordinary differential equation is as follows:

$$y'' = y' + \underline{\text{alf} \cdot y^2}$$

define the function as follows:

void FORTRAN f(double *x, double *y, int *m, double *alf)
{
y[2] = y[1] + (*alf) * y[0] * y[0];
}

The Input arguments: m=2, isw=1.

(c) If the boundary conditions of the high-order ordinary differential equation are given by:

$$\text{left-hand side boundary} \begin{cases} y^{(b_1)} = c_1 \\ y^{(b_2)} = c_2 \\ \vdots \\ y^{(b_p)} = c_p \end{cases} \quad \text{right-hand side boundary} \begin{cases} y^{(b_{p+1})} = c_{p+1} \\ y^{(b_{p+2})} = c_{p+2} \\ \vdots \\ y^{(b_m)} = c_m \end{cases}$$

set array in, ic and bn as follows:
in[i − 1] = 0 (i = 1, 2, $\cdots$, p)
in[i − 1] = 1 (i = p + 1, p + 2, $\cdots$, m)
ic[i − 1] = b_i(i = 1, 2, $\cdots$, m)
bn[i − 1] = c_i(i = 1, 2, $\cdots$, m)
Example:
If the boundary conditions are given by:

left-hand side boundary $y = 1.0$ right-hand side boundary $y' = 2.0$

the input values of in, ic and bn are as follows:
in[0]=0, ic[0]=0, bn[0]=1.0
in[1]=1, ic[1]=1, bn[1]=2.0

(7) **Example**

(a) Problem

Solve the following ordinary differential equation:

$$y'' = y' + y^2$$

under the following boundary conditions:

$$y|_{x=1.0} = 1.0, y'|_{x=2.0} = 2.0$$

(b) Input data

Name of function f(x, y, m, alf): f, xa=1.0, xb=2.0,
in[0]=0, ic[0]=0, bn[0]=1.0,
in[1]=1, ic[1]=1, bn[1]=2.0,
m=2, x, k=6, er, ea, nx, nev=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dohnnv */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *y,int *m,double *alf)
#else
void    f(x,y,m,alf)
double *x;
double *y;
double *alf;
int    *m;
#endif
{
    y[2]= y[1]+(*alf)*(y[0]*y[0]);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double ax;
    double bx;
    int *in;
    int *ic;
    double *bn;
    int m;
    double *x;
    int k;
    double epr;
    double epa;
    int nx;
    int nev;
    double *y;
    int isw;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,nwk;
    FILE *fp;

    fp = fopen( "dohnnv.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dohnnv ***\n" );
    printf( "\n     ** Input **\n\n" );
    m=2;
    k=6;

    in  = ( int * )malloc((size_t)( sizeof(int) * m ));
    if( in  == NULL )
    {
        printf( "no enough memory for array in\n" );
```

```
    return -1;
}

ic  = ( int * )malloc((size_t)( sizeof(int) * m ));
if( ic  == NULL )
{
    printf( "no enough memory for array ic\n" );
    return -1;
}

bn  = ( double * )malloc((size_t)( sizeof(double) * m ));
if( bn  == NULL )
{
    printf( "no enough memory for array  bn\n" );
    return -1;
}
x  = ( double * )malloc((size_t)( sizeof(double) * k ));
if( x  == NULL )
{
    printf( "no enough memory for array  x\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * (k*(m+1)) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * m ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

fscanf( fp, "%lf", &ax );
fscanf( fp, "%lf", &bx );
printf( "\txa  = %8.3g\n", ax );
printf( "\txb  = %8.3g\n", bx );
printf( "\n\tBoundary Condition\n\n" );
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%d", &in[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%d", &ic[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &bn[i] );
}
printf( "\t  Position   Order of y       Value\n" );
for( i=0 ; i<m ; i++ )
{
    printf( "\t%6d     %6d         %8.3g\n",in[i],ic[i],bn[i]);
}
printf( "\n\tOrder of Differential Equation = %6d\n",m);
printf( "\n\tPoints Where Approximate Values are Computed\n\n" );
for( i=0 ; i<k ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
    printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
}
printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",k );

fscanf( fp, "%lf", &epr );
fscanf( fp, "%lf", &epa );
fscanf( fp, "%d", &nx );
fscanf( fp, "%d", &nev );
fscanf( fp, "%d", &isw );

nwk=(m+1)*(m+1)*(nx+1)+m*( (2*m+1>16) ? 2*m+1 : 16 );
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}


printf( "\tRequired Local Relative Precision = %8.3g\n", epr );
printf( "\tRequired Local Absolute Precision = %8.3g\n", epa);
printf( "\tMaximum Number of Shooting Points =   %6d\n", nx );
printf( "\tMaximum Iterative Number = %6d\n", nev );
printf( "\tisw = %6d\n", isw );
printf( "\n" );
```

```
    fclose( fp );

    ierr = ASL_dohnnv(f, ax, bx, in, ic, bn, m, x, k, epr, epa, &nx, &nev, y, isw, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tPractical Number of Shooting Points = %6d\n", nx );
    printf( "\tPractical Iterative Number = %6d\n", nev );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        for( j=0 ; j<m+1 ; j++ )
        {
        printf( "\t  y[%6d][%6d] = %8.3g\n",i,j,y[i+k*j] );
        }
    }

    free( in );
    free( ic );
    free( bn );
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dohnnv ***

 ** Input **

xa  =        1
xb  =        2

Boundary Condition

  Position   Order of y       Value
     0          0               1
     1          1               2

Order of Differential Equation =      2

Points Where Approximate Values are Computed

  x[     0] =        1
  x[     1] =      1.2
  x[     2] =      1.4
  x[     3] =      1.6
  x[     4] =      1.8
  x[     5] =        2

Number of Points Where Approximate Values are Computed =      6
Required Local Relative Precision =        0
Required Local Absolute Precision =        0
Maximum Number of Shooting Points =       10
Maximum Iterative Number =      0
isw =      1


 ** Output **

ierr =      0

Practical Number of Shooting Points =      6
Practical Iterative Number =     12

Approximate Values

  y[     0][     0] =        1
  y[     0][     1] =  -0.0831
  y[     0][     2] =    0.917
  y[     1][     0] =        1
  y[     1][     1] =    0.119
  y[     1][     2] =     1.12
  y[     2][     0] =     1.05
  y[     2][     1] =    0.377
  y[     2][     2] =     1.48
  y[     3][     0] =     1.16
  y[     3][     1] =    0.727
  y[     3][     2] =     2.07
  y[     4][     0] =     1.35
  y[     4][     1] =     1.23
  y[     4][     2] =     3.06
  y[     5][     0] =     1.67
  y[     5][     1] =        2
  y[     5][     2] =     4.79
```

2.3.6 ASL_dohnnf, ASL_rohnnf
High-Order Ordinary Differential Equation (Function Boundary Conditions)

(1) **Function**

ASL_dohnnf or ASL_rohnnf uses the multipoint shooting method to solve a boundary value problem for a high-order ordinary differential equation for which boundary conditions are given as functions.

(2) **Usage**

Double precision:

```
ierr = ASL_dohnnf  (f, fb, xa, xb, m, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rohnnf  (f, fb, xa, xb, m, x, k, er, ea, &nx, &nev, y, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, y, m, alf) that defines the differential equation as a function of x and y.
2	fb	–	–	Input	Name of function fb(ya, yb, m, g) that defines boundary conditions.
3	xa	{D / R}	1	Input	x coordinate of left-hand side boundary
4	xb	{D / R}	1	Input	x coordinate of right-hand side boundary
5	m	I	1	Input	Maximum differential order of variable y in differential equations
6	x	{D∗ / R∗}	k	Input	x coordinate x_i where approximate solutions are calculated
7	k	I	1	Input	Number of calculation points
8	er	{D / R}	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
9	ea	{D / R}	1	Input	Required local absolute precision (Default value: Unit for determining error)

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	nx	I*	1	Input	Maximum number of shooting points
				Output	Actual number of shooting points
11	nev	I*	1	Input	Maximum number of iterations (Default value: 100)
				Output	Actual number of iterations
12	y	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	k×(m + 1)	Output	Solutions $y^{(v)}(x_i)$ y[(i − 1) + k × v] = $y^{(v)}(x_i)$
13	isw	I	1	Input	Parameterization processing switch 0: Parameterization not performed Nonzero: Parameterization performed
14	iwk	I*	m	Work	Work area
15	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $(\text{m}+1)^2 \times (\text{nx}+1) + \text{m} \times \max(2 \times \text{m} + 1, 16)$
16	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa < xb

(b) er≥ e_r. Where, e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(c) ea ≥ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) nev > 0 (Except when 0 is entered in order to set nev to the default value)

(e) m ≥ 1

(f) k ≥ 1

(g) xa ≤ x[i − 1] ≤ xb (i = 1, 2, ⋯ , k)

(h) nx ≥ min(5i + 1, 51)
Where, i is minimum integer for which xb − xa ≤ i

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where xa ≠ xb)	Processing is performed assuming xa and xb are replaced each other.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with the corresponding default value set.

ierr value	Meaning	Processing
3000	xa=xb	Processing is aborted.
3010	Restriction (e) was not satisfied.	
3020	Restriction (f) was not satisfied.	
3030	Restriction (g) was not satisfied.	
3040	Restriction (h) was not satisfied.	
4000	The shooting point could not be added due to severe oscillation (large derivative), etc.	
4500	The step size became too small during the initial value problem calculation (existence of singularity, etc.)	
5000	The maximum number of shooting points was reached.	
5500	The maximum number of iterations was reached (including cases when solution does not exist or is indefinite).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) In a nonlinear problem for a high-order ordinary differential equation expressed as follows:

$$y^{(m)} = f(x, y, y', \cdots, y^{(j)}, \cdots, y^{(m-1)})$$

(Where, differential order j of right-hand side $y^{(j)}$ must satisfy $j \leq m-1$.) if there is a multiplication or division of two or more of the variables $y^{(j)}$ in the terms on the right-hand side of the equation such as $y \cdot y'$, $\frac{y}{y'}$, y^2, or a function other than a sum or scalar multiple of $y^{(j)}$ such as sin(y) or abs(y'), parameterize this nonlinear problem by multiplying this portion by alf.
Example :

$$y'' = y' + y^2$$

Parameterize this nonlinear problem as follows:

$$y'' = y' + \underline{\text{alf} \cdot y^2}$$

When the problem has been parameterized, set isw = 1.
A linear problem need not be parameterized. In this case, set isw = 0.

(b) The actual name of function f(x, y, m, alf), that defines the differential equations, must be declared in the user program, and the actual function must be created.

The function f(x, y, m, alf) (in double-precision) for the high-order ordinary differential equation that was parameterized as described in Note (a) should be created as follows:

```
void FORTRAN f(double *x, double *y, int *m, double *alf)
{
    y[*m - 1] = f(*alf, x, y, ..., y^(j), ..., y^(m-1))
}
```

where the following correspondences are assumed:
$x \leftrightarrow$ *x, $m \leftrightarrow$ *m, $y \leftrightarrow$ y[0], $y^{(j)} \leftrightarrow$ y[j]
$f(*\text{alf}, x, \cdots)$ is parameterized one for $f(x, \cdots)$.

Example:
When parameterized ordinary differential equation is as follows:

$$y'' = y' + \underline{\text{alf} \cdot y^2}$$

define the function as follows:

```
void FORTRAN f(double *x, double *y, int *m, double *alf)
{
    y[2] = y[1] + (*alf) * y[0] * y[0];
}
```

The Input arguments: m=2, isw=1.

(c) The actual name of function fb(ya, yb, m, g), that defines the boundary conditions, must be declared in the user program, and the actual function must be created.

When m boundary conditions are given as follows using values $y^{(j)}(a)$ of $y^{(j)}$ at left-hand side boundary $x = a$ and values $y^{(j)}(b)$ of $y^{(j)}$ at right-hand side boundary $x = b$:

$$\begin{cases} g_1(y(a), \cdots, y^{(j)}(a), \cdots, y^{(m-1)}(a), y(b), \cdots, y^{(j)}(b), \cdots, y^{(m-1)}(b)) = 0 \\ g_2(y(a), \cdots, y^{(j)}(a), \cdots, y^{(m-1)}(a), y(b), \cdots, y^{(j)}(b), \cdots, y^{(m-1)}(b)) = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ g_m(y(a), \cdots, y^{(j)}(a), \cdots, y^{(m-1)}(a), y(b), \cdots, y^{(j)}(b), \cdots, y^{(m-1)}(b)) = 0 \end{cases}$$

the function fb(ya, yb, m, g) (in double-precision) should be created as follows:

```
void FORTRAN fb(double *ya, double *yb, int *m, double *g)
{
    g[0] = g1(y(a), ..., y^(j)(a), ..., y^(m-1)(a), y(b), ..., y^(j)(b), ..., y^(m-1)(b));
    g[1] = g2(y(a), ..., y^(j)(a), ..., y^(m-1)(a), y(b), ..., y^(j)(b), ..., y^(m-1)(b));
      :
    g[m - 1] = gm(y(a), ..., y^(j)(a), ..., y^(m-1)(a), y(b), ..., y^(j)(b), ..., y^(m-1)(b));
}
```

where the following correspondences are assumed.
$m \leftrightarrow$*m, $y(a) \leftrightarrow$ ya[0], $y^{(j)}(a) \leftrightarrow$ ya[j]
$y(b) \leftrightarrow$ yb[0], $y^{(j)}(b) \leftrightarrow$ yb[j]
Example:
When boundary conditions are given as follows:

$$\begin{cases} y(a) - y(b) = 0.0 \\ y'(b) = 1.0 \end{cases}$$

define the function as follows:

```
void FORTRAN fb(double *ya, double *yb, int *m, double *g)
{
    g[0] = ya[0] - yb[0];
    g[1] = yb[1] - 1.0;
}
```

(7) **Example**

(a) Problem

Solve the following ordinary differential equation:

$$y'' = y' + y^2$$

under the following boundary conditions:

$$\begin{cases} y(1.0) - y(2.0) = 0.0 \\ y(2.0) = 2.0 \end{cases}$$

(b) Input data

Name of function f(x, y, m, alf): f1 and name of function fb(ya, yb, m, g): f2, xa=1.0, xb=2.0, m=2, x, k=3, er, ea, nx, nev=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dohnnf */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f1(double *x,double *y,int *m,double *alf)
#else
void    f1(x,y,m,alf)
double *x;
double *y;
double *alf;
int    *m;
#endif
{
    y[2]= y[1]+(*alf)*(y[0]*y[0]);
}
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f2(double *ya,double *yb,int *m,double *g)
#else
void    f2(ya,yb,m,g)
double *ya;
double *yb;
double *g;
int    *m;
#endif
{
    g[0]= ya[0]-yb[0];
    g[1]= yb[0]-2.0;
}
#ifdef __cplusplus
}
#endif


int main()
{
    double ax;
    double bx;
    int m;
    double *x;
    int k;
    double epr;
    double epa;
    int nx;
    int nev;
    double *y;
    int isw;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,nwk;
    FILE *fp;
```

```
fp = fopen( "dohnnf.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dohnnf ***\n" );
printf( "\n     ** Input **\n\n" );
m=2;
k=3;

x  = ( double * )malloc((size_t)( sizeof(double) * k ));
if( x  == NULL )
{
    printf( "no enough memory for array  x\n" );
    return -1;
}

y  = ( double * )malloc((size_t)( sizeof(double) * (k*(m+1)) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * m ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

fscanf( fp, "%lf", &ax );
fscanf( fp, "%lf", &bx );
printf( "\txa  =%8.3g\n", ax );
printf( "\txb  =%8.3g\n", bx );
printf( "\n\tBoundary Conditions\n" );
printf( "\t  ya-yb = 0.0 \n" );
printf( "\t  yb    = 2.0    (ya=y(x=1.0),yb=y(x=2.0))\n" );
printf( "\n\tOrder of Differential Equation = %6d\n",m);
printf( "\n\tPoints Where Approximate Values are Computed\n\n" );
for( i=0 ; i<k ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
    printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
}
printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",k );

fscanf( fp, "%lf", &epr );
fscanf( fp, "%lf", &epa );
fscanf( fp, "%d", &nx );
fscanf( fp, "%d", &nev );
fscanf( fp, "%d", &isw );

nwk=(m+1)*(m+1)*(nx+1)+m*( (2*m+1>16) ? 2*m+1 : 16 );
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}


printf( "\n\tRequired Local Relative Precision = %8.3g\n", epr );
printf( "\tRequired Local Absolute Precision = %8.3g\n", epa);
printf( "\tMaximum Number of Shooting Points =   %6d\n", nx );
printf( "\tMaximum Iterative Number = %6d\n", nev );
printf( "\tisw = %6d\n", isw );

fclose( fp );

ierr = ASL_dohnnf(f1, f2, ax, bx, m, x, k, epr, epa, &nx, &nev, y, isw, iwk, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n\tPractical Number of Shooting Points = %6d\n", nx );
printf( "\tPractical Iterative Number = %6d\n", nev );
printf( "\n\tApproximate Values\n\n" );
for( i=0 ; i<k ; i++ )
{
    for( j=0 ; j<m+1 ; j++ )
    {
    printf( "\t  y[%6d][%6d] = %8.3g\n",i,j,y[i+k*j] );
    }
}
```

```
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dohnnf ***

 ** Input **

xa  =       1
xb  =       2

Boundary Conditions
  ya-yb = 0.0
  yb    = 2.0    (ya=y(x=1.0),yb=y(x=2.0))

Order of Differential Equation =      2

Points Where Approximate Values are Computed

  x[     0] =        1
  x[     1] =      1.5
  x[     2] =        2

Number of Points Where Approximate Values are Computed =      3

Required Local Relative Precision =        0
Required Local Absolute Precision =        0
Maximum Number of Shooting Points =       50
Maximum Iterative Number =      0
isw =      1

 ** Output **

ierr =      0

Practical Number of Shooting Points =      6
Practical Iterative Number =     11

Approximate Values

  y[     0][     0] =        2
  y[     0][     1] =    -1.32
  y[     0][     2] =     2.68
  y[     1][     0] =     1.64
  y[     1][     1] =   -0.104
  y[     1][     2] =      2.6
  y[     2][     0] =        2
  y[     2][     1] =     1.78
  y[     2][     2] =     5.78
```

2.3.7 ASL_dohnlv, ASL_rohnlv
High-Order Linear Ordinary Differential Equation

(1) **Function**

ASL_dohnlv or ASL_rohnlv uses the collocation method to solve a boundary value problem for a high-order linear ordinary differential equation for which boundary conditions are given as input values.

(2) **Usage**

Double precision:

```
ierr = ASL_dohnlv  (f, xa, xb, in, ic, bn, m, x, k, er, ea, &nx, y, iwk, wk);
```

Single precision:

```
ierr = ASL_rohnlv  (f, xa, xb, in, ic, bn, m, x, k, er, ea, &nx, y, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, al, m) that defines the coefficients and constant term of the high-order linear ordinary differential equation.
2	xa	{D / R}	1	Input	x coordinate of left-hand side boundary
3	xb	{D / R}	1	Input	x coordinate of right-hand side boundary
4	in	I*	m	Input	Positions where boundary conditions are set (xa side: 0, xb side: Nonzero)
5	ic	I*	m	Input	Differential order j of variable $y^{(j)}$ which gives boundary conditions value
6	bn	{D∗ / R∗}	m	Input	Boundary condition values given for variable $y^{(j)}$
7	m	I	1	Input	Maximum differential order of variable y in differential equations
8	x	{D∗ / R∗}	k	Input	x coordinate x_i where approximate solutions are calculated
9	k	I	1	Input	Number of calculation points
10	er	{D / R}	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
11	ea	{D / R}	1	Input	Required local absolute precision (Default value: Unit for determining error)
12	nx	I*	1	Input	Maximum number of selected points
				Output	Actual number of selected points
13	y	{D∗ / R∗}	k×(m + 1)	Output	Solutions $y^{(v)}(x_i)$ y[(i − 1) + k × v] = $y^{(v)}(x_i)$
14	iwk	I*	3 × m + nx	Work	Work area
15	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: 2 × nx × nx + 5 × nx + 2 × m + 2
16	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa < xb

(b) er$\geq e_r$. Where, e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(c) ea $\geq$ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) M ≥ 0

(e) $0 \leq$ ic[i − 1] < m (i $= 1, 2, \cdots,$ m)

(f) k ≥ 1

(g) xa $\leq$ x[i − 1] $\leq$ xb (i $= 1, 2, \cdots,$ k)

(h) nx $\geq \max(\min(5 \times \text{i} + 1, 101), \text{m} + 2)$
Where, i is minimum integer for which xb − xa $\leq$ i

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where, xa ≠ xb).	Processing is performed assuming that xa and xb are replaced each other.
1500	Restriction (b) or (c) was not satisfied.	Processing is performed with the corresponding default value set.
3000	xa=xb	Processing is aborted.
3010	Restriction (d) was not satisfied.	
3020	Restriction (e) was not satisfied.	
3030	Restriction (f) was not satisfied.	
3040	Restriction (g) was not satisfied.	
3050	Restriction (h) was not satisfied.	
4000	The simultaneous first-order equations could not be solved.	
5000	The maximum number of selected points was reached (including the case when there is no solution).	The solution at that time is returned, and processing is aborted.

(6) **Notes**

(a) The actual name of function f(x, al, m), that defines the differential equation, must be declared in the user program, and the actual function must be created.

For the following high-order linear ordinary differential equation:

$$f_1(x)y^{(m)} + f_2(x)y^{(m-1)} + \cdots + f_{m+1}(x)y + f_{m+2}(x) = 0$$

the function f(x, al, m) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *al, int *m)
{
    al[0] = f1(x);
    al[1] = f2(x);
      ⋮
    al[m + 1] = fm+2(x);
}
```

where the following correspondences are assumed:

$x \leftrightarrow *$x, $m \leftrightarrow *$m

Example:

$$y'' + 2xy + x = 0$$

```
void FORTRAN f(double *x, double *al, int *m)
{
    al[0] = 1.0;
    al[1] = 0.0;
    al[2] = 2 * (*x);
    al[3] = (*x);
}
```

The Input argument:m=2.

(b) If the boundary conditions of the high-order linear ordinary differential equation are given by:

$$\text{left-hand side boundary} \begin{cases} y^{(b_1)} = c_1 \\ y^{(b_2)} = c_2 \\ \quad\vdots \\ y^{(b_p)} = c_p \end{cases} \quad \text{right-hand side boundary} \begin{cases} y^{(b_{p+1})} = c_{p+1} \\ y^{(b_{p+2})} = c_{p+2} \\ \quad\vdots \\ y^{(b_m)} = c_m \end{cases}$$

set array in, ic and bn as follows:

in[i − 1] = 0 (i = 1, 2, ⋯ , p)

in[i − 1] = 1 (i = p + 1, p + 2, ⋯ , m)

ic[i − 1] = b_i(i = 1, 2, ⋯ , m)

bn[i − 1] = c_i(i = 1, 2, ⋯ , m)

Example: If the boundary conditions are given by:

$$\text{left-hand side boundary} \quad y = 1.0 \quad \text{right-hand side boundary} \quad y' = 2.0$$

the input values of in, ic and bn are as follows:

in[0]=0, ic[0]=0, bn[0]=1.0

in[1]=1, ic[1]=1, bn[1]=2.0

(7) **Example**

(a) Problem

Solve the following ordinary differential equation:

$$y'' + 2xy + x = 0$$

under the following boundary conditions:

$$y|_{x=0.0113} = 10.0, y'|_{x=0.0188} = 12.0$$

(b) Input data

Name of function f(x, al, m):f, xa=0.0113, xb=0.0188,
in[0]=0, ic[0]=0, bn[0]=10.0,
in[1]=1, ic[1]=1, bn[1]=20.0,
m=2, x, k=6, er, ea and nx.

(c) Main program

```
/*      C interface example for ASL_dohnlv */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *al,int *m)
#else
void    f(x,al,m)
double *x;
double *al;
int    *m;
#endif
{
    al[0]= 1.0;
    al[1]= 0.0;
    al[2]= 2.0*(*x);
    al[3]= (*x);
}
#ifdef __cplusplus
}
#endif


int main()
{
    double ax;
    double bx;
    int *in;
    int *ic;
    double *bn;
    int m;
    double *x;
    int k;
    double epr;
    double epa;
    int nx;
    double *y;
    int *iwk;
    double *wk;
    int ierr;
    int i,j,nwk;
    FILE *fp;

    fp = fopen( "dohnlv.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dohnlv ***\n" );
    printf( "\n     ** Input **\n\n" );
    m=2;
    k=6;

    in  = ( int * )malloc((size_t)( sizeof(int) * m ));
    if( in  == NULL )
    {
```

```
    printf( "no enough memory for array in\n" );
    return -1;
}

ic  = ( int * )malloc((size_t)( sizeof(int) * m ));
if( ic  == NULL )
{
    printf( "no enough memory for array ic\n" );
    return -1;
}

bn  = ( double * )malloc((size_t)( sizeof(double) * m ));
if( bn  == NULL )
{
    printf( "no enough memory for array  bn\n" );
    return -1;
}
x  = ( double * )malloc((size_t)( sizeof(double) * k ));
if( x  == NULL )
{
    printf( "no enough memory for array  x\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * (k*(m+1)) ));
if( y  == NULL )
{
    printf( "no enough memory for array y \n" );
    return -1;
}

fscanf( fp, "%lf", &ax );
fscanf( fp, "%lf", &bx );
printf( "\txa  = %8.3g\n", ax );
printf( "\txb  = %8.3g\n", bx );
printf( "\n\tBoundary Condition\n\n" );
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%d", &in[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%d", &ic[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &bn[i] );
}
printf( "\t  Position   Order of y       Value\n" );
for( i=0 ; i<m ; i++ )
{
    printf( "\t%6d     %6d         %8.3g\n",in[i],ic[i],bn[i]);
}
printf( "\n\tOrder of Differential Equation = %6d\n",m);
printf( "\n\tPoints Where Approximate Values are Computed\n\n" );
for( i=0 ; i<k ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
    printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
}
printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",k );

fscanf( fp, "%lf", &epr );
fscanf( fp, "%lf", &epa );
fscanf( fp, "%d", &nx );

printf( "\n\tRequired Local Relative Precision  = %8.3g\n", epr );
printf( "\tRequired Local Absolute Precision  = %8.3g\n", epa);
printf( "\tMaximum Number of Selected Points  =   %6d\n", nx );

nwk=2*nx*nx+5*nx+2*m+2;
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk \n" );
    return -1;
}

iwk  = ( int * )malloc((size_t)( sizeof(int) * (3*m+nx) ));
if( iwk  == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

fclose( fp );

ierr = ASL_dohnlv(f, ax, bx, in, ic, bn, m, x, k, epr, epa, &nx, y, iwk, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
```

```
    printf( "\n\tPractical Number of Selected Points = %6d\n", nx );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        for( j=0 ; j<m+1 ; j++ )
        {
        printf( "\t  y[%6d][%6d] = %8.3g\n",i,j,y[i+k*j] );
        }
    }

    free( in );
    free( ic );
    free( bn );
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dohnlv ***

 ** Input **

xa  =     0.01
xb  =     0.02

Boundary Condition

  Position   Order of y       Value
     0          0               10
     1          0               12

Order of Differential Equation =      2

Points Where Approximate Values are Computed

  x[     0] =   0.0113
  x[     1] =   0.0143
  x[     2] =   0.0167
  x[     3] =   0.0171
  x[     4] =   0.0183
  x[     5] =   0.0188

Number of Points Where Approximate Values are Computed =      6

Required Local Relative Precision  =    1e-08
Required Local Absolute Precision  =    1e-08
Maximum Number of Selected Points  =      100

 ** Output **

ierr =      0

Practical Number of Selected Points =      6

Approximate Values

  y[     0][     0] =     10.3
  y[     0][     1] =      200
  y[     0][     2] =   -0.243
  y[     1][     0] =     10.9
  y[     1][     1] =      200
  y[     1][     2] =   -0.325
  y[     2][     0] =     11.3
  y[     2][     1] =      200
  y[     2][     2] =   -0.395
  y[     3][     0] =     11.4
  y[     3][     1] =      200
  y[     3][     2] =   -0.408
  y[     4][     0] =     11.7
  y[     4][     1] =      200
  y[     4][     2] =   -0.445
  y[     5][     0] =     11.8
  y[     5][     1] =      200
  y[     5][     2] =   -0.461
```

2.3.8 ASL_dolnlv, ASL_rolnlv
Second-Order Linear Ordinary Differential Equation

(1) **Function**

ASL_dolnlv or ASL_rolnlv uses the coefficient determination method to solve a boundary value problem for a second-order linear ordinary differential equation for which boundary conditions are given as input values.

(2) **Usage**

Double precision:

ierr = ASL_dolnlv (f, xa, xb, in, ic, bn, x, k, er, ea, y, wk);

Single precision:

ierr = ASL_rolnlv (f, xa, xb, in, ic, bn, x, k, er, ea, y, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	–	–	Input	Name of function f(x, al) that defines the coefficients and constant term of the second-order linear ordinary differential equation.
2	xa	{D / R}	1	Input	x coordinate of left-hand side boundary
3	xb	{D / R}	1	Input	x coordinate of right-hand side boundary
4	in	I*	2	Input	Positions where boundary conditions are set (xa side: 0, xb side: Nonzero)
5	ic	I*	2	Input	Differential order j of variable $y^{(j)}$ which gives boundary conditions value
6	bn	{D*/ R*}	2	Input	Boundary condition values given for variable $y^{(j)}$
7	x	{D*/ R*}	k	Input	x coordinate x_i where approximate solutions are calculated
8	k	I	1	Input	Number of calculation points
9	er	{D / R}	1	Input	Required local relative precision **Default value**: Double precision : 10^{-12} Single precision : 10^{-5}
10	ea	{D / R}	1	Input	Required local absolute precision (Default value: Unit for determining error)
11	y	{D*/ R*}	k × 3	Output	Solutions $y^{(v)}(x_i)$ y[(i − 1) + k × v] = $y^{(v)}(x_i)$
12	wk	{D*/ R*}	6 × k + 35	Work	Work area
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa < xb

(b) er≥ e_r. Where, e_r= double precision:10^{-14}, single precision: 10^{-5} (Except when 0.0 is entered in order to set er to the default value)

(c) ea ≥ (Minimum expressible absolute value) $\times 2^{24}$
(Except when 0.0 is entered in order to set ea to the default value)

(d) $0 \leq \mathrm{ic}[\mathrm{i}-1] < 2$ $(\mathrm{i} = 1, 2)$

(e) $\mathrm{k} \geq 1$

(f) $\mathrm{xa} \leq \mathrm{x}[\mathrm{i}-1] \leq \mathrm{xb}$ $(\mathrm{i} = 1, 2, \cdots, \mathrm{k})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied. (where, xa ≠ xb).	Processing is performed assuming that xa and xb are replaced each other.
1500	Restriction (b) or (c) was not satisfied.	Processing is performed with the corresponding default value set.
3000	xa=xb	Processing is aborted.
3010	Restriction (d) was not satisfied.	
3020	Restriction (e) was not satisfied.	
3030	Restriction (f) was not satisfied.	
4000	The step size became too small during the initial value problem calculation.	
4500	There is no solution or the solution is indefinite.	

(6) **Notes**

(a) The actual name of function f(x, al), that defines the differential equations, must be declared in the user program, and the actual function must be created.

For the following second-order linear ordinary differential equation:

$$f_1(x)y'' + f_2(x)y' + f_3(x)y + f_4(x) = 0$$

the function f(x, al) (in double-precision) should be created as follows:

```
void FORTRAN f(double *x, double *al)
{
    al[0] = f1(x);
    al[1] = f2(x);
    al[2] = f3(x);
    al[3] = f4(x);
}
```

where the following correspondence is assumed.

$x \leftrightarrow$ *x

Example:

$$y'' + 2xy + x = 0$$

```
void FORTRAN f(double *x, double *al)
{
    al[0] = 1.0;
    al[1] = 0.0;
    al[2] = 2 * (*x);
    al[3] = (*x);
}
```

(b) If the boundary conditions of the second-order ordinary differential equation are given by:

$$\text{left-hand side boundary} \quad y^{(b_1)} = c_1 \quad \text{right-hand side boundary} \quad y^{(b_2)} = c_2$$

set array in, ic and bn as follows:
in[0] = 0
in[1] = 1
ic[i − 1] = b_i (i = 1, 2)
bn[i − 1] = c_i (i = 1, 2)
Example: If the boundary conditions are given by:

$$\text{left-hand side boundary} \quad y = 1.0 \quad \text{right-hand side boundary} \quad y' = 2.0$$

the input values of in, ic and bn are as follows:
in[0]=0, ic[0]=0, bn[0]=1.0
in[1]=1, ic[1]=1, bn[1]=2.0

(7) **Example**

(a) Problem
Solve the following ordinary differential equation:

$$y'' + 2xy + x = 0$$

under the following boundary conditions:

$$y|_{x=0.0} = 1.0, y'|_{x=1.0} = 2.0$$

(b) Input data
Name of function f(x, al):f, xa=0.0, xb=1.0,
in[0]=0, ic[0]=0, bn[0]=1.0,
in[1]=1, ic[1]=1, bn[1]=2.0,
x, k=6, er and ea.

(c) Main program

```
/*      C interface example for ASL_dolnlv */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    f(double *x,double *al)
#else
void    f(x,al)
```

```
double *x;
double *al;
#endif
{
    al[0]= 1.0;
    al[1]= 0.0;
    al[2]= 2.0*(*x);
    al[3]= (*x);
}
#ifdef __cplusplus
}
#endif


int main()
{
    double ax;
    double bx;
    int in[2];
    int ic[2];
    double bn[2];
    double *x;
    int k;
    double epr;
    double epa;
    double *y;
    double *wk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dolnlv.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dolnlv ***\n" );
    printf( "\n     ** Input **\n\n" );
    k=6;

    x  = ( double * )malloc((size_t)( sizeof(double) * k ));
    if( x  == NULL )
    {
        printf( "no enough memory for array  x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * (3*k) ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y \n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (6*k+35) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk \n" );
        return -1;
    }

    fscanf( fp, "%lf", &ax );
    fscanf( fp, "%lf", &bx );
    printf( "\txa  = %8.3g\n", ax );
    printf( "\txb  = %8.3g\n", bx );
    printf( "\n\tBoundary Condition\n\n" );
    for( i=0 ; i<2 ; i++ )
    {
        fscanf( fp, "%d", &in[i] );
    }
    for( i=0 ; i<2 ; i++ )
    {
        fscanf( fp, "%d", &ic[i] );
    }
    for( i=0 ; i<2 ; i++ )
    {
        fscanf( fp, "%lf", &bn[i] );
    }
    printf( "\t  Position   Order of y       Value\n" );
    for( i=0 ; i<2 ; i++ )
    {
        printf( "\t%6d     %6d         %8.3g\n",in[i],ic[i],bn[i]);
    }
    printf( "\n\tOrder of Differential Equation = 2\n");
    printf( "\n\tPoints Where Approximate Values are Computed\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
        printf( "\t  x[%6d] = %8.3g\n",i,x[i]);
```

```
    }
    printf( "\n\tNumber of Points Where Approximate Values are Computed = %6d\n",k );

    fscanf( fp, "%lf", &epr );
    fscanf( fp, "%lf", &epa );

    printf( "\tRequired Local Relative Precision = %8.3g\n", epr );
    printf( "\tRequired Local Absolute Precision = %8.3g\n", epa);

    fclose( fp );

    ierr = ASL_dolnlv(f, ax, bx, in, ic, bn, x, k, epr, epa, y, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tApproximate Values\n\n" );
    for( i=0 ; i<k ; i++ )
    {
        for( j=0 ; j<3 ; j++ )
        {
        printf( "\t  y[%6d][%6d] = %8.3g\n",i,j,y[i+k*j] );
        }
    }

    free( x );
    free( y );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dolnlv ***

 ** Input **

xa  =        0
xb  =        1

Boundary Condition

  Position   Order of y       Value
     0          0               1
     1          1               2

Order of Differential Equation = 2

Points Where Approximate Values are Computed

  x[     0] =        0
  x[     1] =      0.2
  x[     2] =      0.4
  x[     3] =      0.6
  x[     4] =      0.8
  x[     5] =        1

Number of Points Where Approximate Values are Computed =      6
Required Local Relative Precision =        0
Required Local Absolute Precision =        0

 ** Output **

ierr =      0

Approximate Values

  y[     0][     0] =        1
  y[     0][     1] =     8.54
  y[     0][     2] =        0
  y[     1][     0] =      2.7
  y[     1][     1] =     8.44
  y[     1][     2] =    -1.28
  y[     2][     0] =     4.35
  y[     2][     1] =     7.94
  y[     2][     2] =    -3.88
  y[     3][     0] =     5.84
  y[     3][     1] =     6.81
  y[     3][     2] =    -7.61
  y[     4][     0] =     7.02
  y[     4][     1] =     4.85
  y[     4][     2] =      -12
  y[     5][     0] =     7.72
  y[     5][     1] =        2
  y[     5][     2] =    -16.4
```

2.4 INTEGRAL EQUATIONS

2.4.1 ASL_doief2, ASL_roief2
Fredholm's Integral Equation of the Second Kind

(1) **Function**

ASL_doief2 or ASL_roief2 uses Gauss' integral formula to solve Fredholm's integral equation of the second kind

$$y(t) - \int_a^b K(t,x)y(x)dx = f(t)$$

to obtain a value $y(t)$ at an arbitrary point $t = t_i$ in the interval

$$-0.4984|a-b| + \frac{a+b}{2} \leq t_i \leq 0.4984|a-b| + \frac{a+b}{2} \quad (i = 1, 2, \cdots, n)$$

according to interpolation using a cubic spline function.

(2) **Usage**

Double precision:

```
ierr = ASL_doief2  (ff, fk, xa, xb, ti, n, y);
```

Single precision:

```
ierr = ASL_roief2  (ff, fk, xa, xb, ti, n, y);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	ff	{D / R}	−	Input	Name of function ff(t) that defines the known function $f(t)$
2	fk	{D / R}	−	Input	Name of function fk(t, x) that defines the regular kernel $K(t,x)$
3	xa	{D / R}	1	Input	Lower bound a of the integration interval
4	xb	{D / R}	1	Input	Upper bound b of the integration interval
5	ti	{D∗ / R∗}	n	Input	Points t_i where approximate solutions are obtained
6	n	I	1	Input	Number of calculation points n

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
7	y	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	n	Output	Approximate solutions $y(t_i)$
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) xa≠xb

(b) n>1

(c) $-0.4984|\text{xa}-\text{xb}|+\dfrac{\text{xa}+\text{xb}}{2} \le \text{ti[i}-1] \le 0.4984|\text{xa}-\text{xb}|+\dfrac{\text{xa}+\text{xb}}{2} \quad (\text{i}=1,\cdots,\text{n})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied.	$\text{y[i}-1] = \text{ff(ti[i}-1])$ $(\text{i}=1,\cdots,\text{n})$
1100	Restriction (c) was not satisfied.	An extrapolated value is output using spline coefficients at the endpoint.
3000	Restriction (b) was not satisfied.	Processing is aborted.
4000	An error occurred when solving the simultaneous linear equations.	

(6) **Notes**

(a) The actual name of function ff(t) and fk(t, x) must be declared in the user program, and the actual function must be created. This function ff(t) and fk(t, x) (in double-precision) should be created as follows:

```
void FORTRAN ff(double *t)
{
    return f(t);
}

void FORTRAN fk(double *t, double *x)
{
    return K(t,x);
}
```

where the following correspondences are assumed.
$t \leftrightarrow$ *t, $x \leftrightarrow$ *x

(7) **Example**

(a) Problem
Solve the following integral equation

$$y(t) - \int_1^2 (x+t)y(x)dx = t$$

for x =1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8 and 1.9.

(b) Input data

Name of function ff(t): ff, name of function fk(t, x): fk, xa=1.0, xb=2.0, ti and n=9.

(c) Main program

```
/*      C interface example for ASL_doief2 */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double ff(double *tt)
#else
double ff(tt)
double *tt;
#endif
{
    return *tt;
}
#ifdef __cplusplus
}
#endif
#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double fk(double *tt,double *x)
#else
double fk(tt,x)
double *tt,*x;
#endif
{
    return (*x) + (*tt);
}
#ifdef __cplusplus
}
#endif
int main()
{
    double xa;
    double xb;
    double *t;
    double *y;
    int n;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "doief2.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_doief2 ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &xa );
    fscanf( fp, "%lf", &xb );
    fscanf( fp, "%d", &n );

    t = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( t == NULL )
    {
        printf( "no enough memory for array t\n" );
        return -1;
    }

    y = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf", &t[i] );
    }

    printf( "\txa   = %8.3g\n", xa );
    printf( "\txb   = %8.3g\n", xb );

    fclose( fp );
```

```
    ierr = ASL_doief2(ff, fk, xa, xb, t, n, y);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tSolution\n" );
    printf( "\n\t i     x[i]     y[i]\n");
    for( i=0 ; i<n ;i++ )
    {
        printf( "\t%2d %8.3g %8.3g \n", i,t[i],y[i]);
    }

    free( t );
    free( y );

    return 0;
}
```

(d) Output results

```
*** ASL_doief2 ***

 ** Input **

xa   =        1
xb   =        2

 ** Output **

ierr =      0

Solution

 i     x[i]     y[i]
 0      1.1   -0.856
 1      1.2   -0.832
 2      1.3   -0.808
 3      1.4   -0.784
 4      1.5    -0.76
 5      1.6   -0.736
 6      1.7   -0.712
 7      1.8   -0.688
 8      1.9   -0.664
```

2.4.2 ASL_doiev1, ASL_roiev1
Volterra's Integral Equation of the First Kind

(1) **Function**

ASL_doiev1 or ASL_roiev1 uses Maclaurin's formula to solve Volterra's integral equation of the first kind

$$\int_a^t K(t,x)y(x)dx = f(t)$$

to obtain a value $y(x)$ at an arbitrary point $x = x_i \ \ (i = 1, 2, \cdots, n)$ according to interpolation using a cubic spline function.

(2) **Usage**

Double precision:

ierr = ASL_doiev1 (ff, fk, xa, xb, m, xi, n, y, w);

Single precision:

ierr = ASL_roiev1 (ff, fk, xa, xb, m, xi, n, y, w);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	ff	{D / R}	−	Input	Name of function ff(t) that defines the known function $f(t)$
2	fk	{D / R}	−	Input	Name of function fk(t, x) that defines the regular kernel $K(t,x)$
3	xa	{D / R}	1	Input	Lower bound a of the integration interval (See Notes (b))
4	xb	{D / R}	1	Input	Upper bound of points where approximate solutions are calculated (See Notes (b))
5	m	I	1	Input	Number of subdivisions of the integration interval (See Notes (b))
6	xi	{D∗ / R∗}	n	Input	Points x_i where approximate solutions are obtained
7	n	I	1	Input	Number of calculation points n
8	y	{D∗ / R∗}	n	Output	Approximate solutions $y(x_i)$
9	w	{D∗ / R∗}	$5 \times \mathrm{m} + 2$	Work	Work area
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n>1

(b) m>1

(c) xa<xb

(d) $\mathrm{xa} + \dfrac{\mathrm{xb} - \mathrm{xa}}{2 \times \mathrm{m}} < \mathrm{xi[i-1]} < \mathrm{xb} \quad (\mathrm{i} = 1, \cdots, \mathrm{n})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (d) was not satisfied.	An extrapolated value is output using spline coefficients at the endpoint.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	

(6) **Notes**

(a) The actual name of function ff(t) and fk(t, x) must be declared in the user program, and the actual function must be created. This function ff(t) and fk(t, x) (in double-precision) should be created as follows:

```
void FORTRAN ff(double *t)
{
    return f(t);
}
void FORTRAN fk(double *t, double *x)
{
    return K(t,x);
}
```

where the following correspondences are assumed.
$t \leftrightarrow$ *t, $x \leftrightarrow$ *x

(b) Use $\dfrac{\mathrm{xb} - \mathrm{xa}}{\mathrm{m}}$ as the step size h for determining the integral calculation interval. (See Section 2.1.2)

(7) **Example**

(a) Problem
Solve the following integral equation

$$t = \int_0^t (1 + t - x) y(x) dx$$

for x =0.025, 0.075, 0.125, 0.175, 0.225, 0.275, 0.325, 0.375, 0.425, 0.475 and 0.500.

(b) Input data
Name of function ff(t): ff, name of function fk(t, x): fk, xa=0.0, xb=0.5, xi, n=11 and m=11.

(c) Main program

```
/*      C interface example for ASL_doiev1 */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double ff(double *tt)
#else
double ff(tt)
double *tt;
#endif
{
    return *tt;
}
#ifdef __cplusplus
}
#endif
#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double fk(double *tt, double *x)
#else
double fk(tt,x)
double *tt,*x;
#endif
{
    return 1 + (*tt) - (*x);
}
#ifdef __cplusplus
}
#endif
int main()
{
    double xa;
    double xb;
    double *t;
    double *y;
    double *w;
    int m;
    int n;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "doiev1.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_doiev1 ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &xa );
    fscanf( fp, "%lf", &xb );
    fscanf( fp, "%d", &m );
    fscanf( fp, "%d", &n );

    t = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( t == NULL )
    {
        printf( "no enough memory for array t\n" );
        return -1;
    }

    y = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    w = ( double * )malloc((size_t)( sizeof(double) * (5*m+2) ));
    if( w == NULL )
    {
        printf( "no enough memory for array w\n" );
        return -1;
    }
    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf", &t[i] );
    }
```

```
        printf( "\txa   = %8.3g\n", xa );
        printf( "\txb   = %8.3g\n", xb );

        fclose( fp );

        ierr = ASL_doiev1(ff, fk, xa, xb, m, t, n, y, w);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tSolution\n" );
        printf( "\n\t i     x[i]     y[i]\n");
        for( i=0 ; i<n ;i++ )
        {
               printf( "\t%2d %8.3g %8.3g \n", i,t[i],y[i]);
        }

        free( t );
        free( y );
        free( w );

        return 0;
    }
```

(d) Output results

```
*** ASL_doiev1 ***

 ** Input **

xa   =        0
xb   =      0.5

 ** Output **

ierr =      0

Solution

 i     x[i]     y[i]
 0    0.025    0.976
 1    0.075    0.928
 2    0.125    0.882
 3    0.175     0.84
 4    0.225    0.799
 5    0.275     0.76
 6    0.325    0.723
 7    0.375    0.687
 8    0.425    0.654
 9    0.475    0.622
10      0.5    0.606
```

2.5 PARTIAL DIFFERENTIAL EQUATIONS

2.5.1 ASL_dopdh2, ASL_ropdh2
Two-Dimensional Inhomogeneous Helmholtz Equation

(1) **Function**

This function uses a two-dimensional five-point difference in a given rectangular area to solve the following inhomogeneous Helmholtz equation.

$$u_{xx} + u_{yy} + \lambda u = f(x, y)$$

(2) **Usage**

Double precision:

ierr = ASL_dopdh2 (dlmd, func, nx, ny, xl, xu, yl, yu, s1, s2, s3, s4, imax, eps, u, imx, isw, iw, w);

Single precision:

ierr = ASL_ropdh2 (dlmd, func, nx, ny, xl, xu, yl, yu, s1, s2, s3, s4, imax, eps, u, imx, isw, iw, w);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	dlmd	{D / R}	1	Input	Value of the coefficient of u in the difference equation λ.
2	func	–	–	Input	Name of the function f(x,y) that defines the differential equation as a function of x and y
3	nx	I	1	Input	Number of division in X-direction n_x
4	ny	I	1	Input	Number of division in Y-direction n_y
5	xl	{D / R}	1	Input	The value of X along the lower point of the domain.
6	xu	{D / R}	1	Input	The value of X along the upper point of the domain.
7	yl	{D / R}	1	Input	The value of Y along the upper point of the domain.
8	yu	{D / R}	1	Input	The value of Y along the upper point of the domain.
9	s1	{D∗ / R∗}	nx+1	Input	Value of Dirichlet boundary condition of bottom edge (when isw[0] $\neq$ 1)When isw[0] = 1, the area is only allocated.
10	s2	{D∗ / R∗}	ny+1	Input	Value of Dirichlet boundary condition of right edge (when isw[1] $\neq$ 1)When isw[1] = 1, the area is only allocated.
11	s3	{D∗ / R∗}	nx+1	Input	Value of Dirichlet boundary condition of top edge (when isw[2] $\neq$ 1)When isw[2] = 1, the area is only allocated.
12	s4	{D∗ / R∗}	ny+1	Input	Value of Dirichlet boundary condition of left edge (when isw[3] $\neq$ 1)When isw[3] = 1, the area is only allocated.

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
13	imax	I	1	Input	Maximum number of iterations when using an iterative method to solve simultaneous linear equations.
14	eps	$\left\{\begin{matrix}\mathrm{D}\\ \mathrm{R}\end{matrix}\right\}$	1	Input	Truncation residual norm $\parallel \boldsymbol{b} - A\boldsymbol{u} \parallel / \parallel \boldsymbol{b} \parallel$ **Specifiable range:** ≥Underflow decision value (See Note (e))
15	u	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Output	Solution vector$u(x_i, y_j)$. $\mathrm{u}(i,j) = u(x_L + i(x_U - x_L)/n_x, y_L + j(y_U - y_L)/n_y)$ **Size**: $(\mathrm{imx}+1) \times (\mathrm{ny}+1)$
16	imx	I	1	Input	Adjustable dimension of array u
17	isw	I*	4	Input	Boundary condition selection switch isw[0] =0: Add Dirichlet condition to bottom edge =1: Add Neumann condition to bottom edge isw[1] =0: Add Dirichlet condition to right edge =1: Add Neumann condition to right edge isw[2] =0: Add Dirichlet condition to top edge =1: Add Neumann condition to top edge isw[3] =0: Add Dirichlet condition to left edge =1: Add Neumann condition to left edge
18	iw	I*	7	Work	Work area
19	w	$\left\{\begin{matrix}\mathrm{D}*\\ \mathrm{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $13 \times (\mathrm{nx}+1) \times (\mathrm{ny}+1) + 5$
20	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) nx ≥ 2, ny ≥ 2

(b) For the boundary conditions to be added to the edges, the Dirichlet condition must be added to at least one edge.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1400	A diagonal element produced by ILU decomposition became less than ϵ times the absolute value of the original diagonal element. (See Note (c))	Replaces diagonal element produced by ILU decomposition with (original diagonal element) , $\times\epsilon$ and processing continues.
2000	Maximum number of iterations has been reached.	Processing continues without performing acceleration.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
4000	The norm of the right-hand side vector is greater than $\sqrt{\text{overflow decision value}}$. (See Notes (d), (e))	
4100	The norm of the residual is greater than $\sqrt{\text{overflow decision value}}$. (See Notes (d), (e))	
4200	$\parallel (\boldsymbol{r}, \boldsymbol{r^*}) \parallel$ is smaller than underflow decision value.	
4210	$\parallel (\boldsymbol{r}, \boldsymbol{r^*}) \parallel$ is greater than overflow decision value.	
4300	$\parallel (A\boldsymbol{p}, \boldsymbol{p^*}) \parallel$ is smaller than underflow decision value.	
4310	$\parallel (A\boldsymbol{p}, \boldsymbol{p^*}) \parallel$ is greater than overflow decision value.	
5000	Any of diagonal element has absolute value smaller than underflow decision value.	

(6) **Notes**

(a) The actual name of function f(x, y), that defines the differential equations, must be declared in the user program, and the actual function must be created. For the high-order simultaneous ordinary differential equations:

$$u_{xx} + u_{yy} + \lambda u = f(x, y)$$

this function f(x, y) (in double-precision) should be created as follows:

```
double FORTRAN func(double *x, double *y)
{
    return f(x,y);
}
```

where the following correspondences are assumed.
$x \leftrightarrow$ *x,$y \leftrightarrow$ *y
Example:

$$f(x, y) = 6x + 6y$$

Therefore, define the function as follows:

```
double FORTRAN func(double *x, double *y)
{
    return 6.0*(*x) + 6.0*(*y);
}
```

(b) For the Dirichlet conditions s1, s2, s3, and s4, nx + 1 (s1, s3) values and ny + 1 (s1, s3) values, excluding values at the vertices of the virtual grid, are stored in arrays sequentially in ascending order of coordinate values.

(c) The singularity prevention constant ϵ is set to 0.1. (See the high-speed function version (4.1.2.2).)

(d) The l^2 norm shown below is used for the norm.

$$\|\boldsymbol{r}\|_2 = \left(\sum_{i=1}^{n} {r_i}^2 \right)^{1/2} \quad \text{where, } \boldsymbol{r} = (r_1, r_2, r_3, \cdots, r_n)^T$$

(e) (Maximum value$\times 10^{-3}$) is set for the overflow decision value and (Minimum positive value$\times 10^{3}$) is set for the underflow decision value.

(7) **Example**

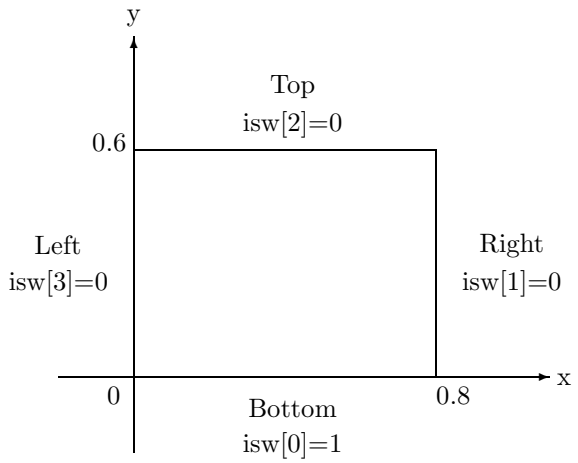
(a) Problem (two-dimensional Poisson equation)
Solve the following two-dimensional Poisson equation :

$$u_{xx} + u_{yy} = f(x, y)$$

under the boundary conditions below.

$$\begin{cases} \frac{\partial u}{\partial y} = 0 & \text{(on the bottom edge)} \\ u = 1 & \text{(on the other edges)} \end{cases}$$

The given two-dimensional area is discretized by an nx × ny orthogonal grid as shown in the following figure.



(b) Input data

dlmd=0.0,

nx=40, ny=50,

xl=0.0, xu=0.6,

yl=0.0, yu=0.8,

imax=200, eps=1.e-10 and

isw=(1, 0, 0, 0).

(c) Main program

```
/*      C interface example for ASL_dopdh2 */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x, double *y)
#else
double f(x,y)
double *x;
double *y;
#endif
{
    return 6.0*(*x) + 6.0*(*y);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double dlmd;
    int idvs;
    int jdvs;
    double x0;
    double x1;
    double y0;
    double y1;
    double *s1;
    double *s2;
    double *s3;
    double *s4;
    int imax;
    double eps;
    double *u;
    int imx=100;
    int *isw;
    int *iwk;
    double *wk;
    int ierr;

    int i,j,nxi,nyi;
    FILE *fp;

    fp = fopen( "dopdh2.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dopdh2 ***\n" );
    printf( "\n     ** Input **\n\n" );

    isw = ( int * )malloc((size_t)( sizeof(int) * 4 ));
    if( isw == NULL )
    {
        printf( "no enough memory for array isw\n" );
        return -1;
    }

    fscanf( fp, "%d", &isw[0]);
    fscanf( fp, "%d", &isw[1]);
    fscanf( fp, "%d", &isw[2]);
    fscanf( fp, "%d", &isw[3]);

    fscanf( fp, "%lf",    &dlmd       );
    fscanf( fp, "%d,%d",  &idvs,&jdvs );
    fscanf( fp, "%lf,%lf",&x0,&x1     );
    fscanf( fp, "%lf,%lf",&y0,&y1     );
```

```
fscanf( fp, "%d",     &imax       );
fscanf( fp, "%lf",    &eps        );

printf( "\tisw   = %6d %6d %6d %6d\n", isw[0],isw[1],isw[2],isw[3] );
printf( "\tdlmd  = %8.3g\n",      dlmd        );
printf( "\tnx,ny =   %6d   %6d\n",   idvs,jdvs);
printf( "\tx0,x1 = %8.3g %8.3g\n",x0,x1       );
printf( "\ty0,y1 = %8.3g %8.3g\n",y0,y1       );
printf( "\timax  = %6d\n",        imax        );
printf( "\teps   = %8.3g\n",      eps         );

s1 = ( double * )malloc((size_t)( sizeof(double) * (idvs+1) ));
if( s1 == NULL )
{
    printf( "no enough memory for array s1\n" );
    return -1;
}

s2 = ( double * )malloc((size_t)( sizeof(double) * (jdvs+1) ));
if( s2 == NULL )
{
    printf( "no enough memory for array s2\n" );
    return -1;
}

s3 = ( double * )malloc((size_t)( sizeof(double) * (idvs+1) ));
if( s3 == NULL )
{
    printf( "no enough memory for array s3\n" );
    return -1;
}

s4 = ( double * )malloc((size_t)( sizeof(double) * (jdvs+1) ));
if( s4 == NULL )
{
    printf( "no enough memory for array s4\n" );
    return -1;
}

u = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(jdvs+1) ));
if( u == NULL )
{
    printf( "no enough memory for array u\n" );
    return -1;
}

iwk = ( int * )malloc((size_t)( sizeof(int) * 7 ));
if( iwk == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

wk  = ( double * )malloc((size_t)( sizeof(double) * ((imx+1)*(jdvs+1)*13+5) ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}
fclose( fp );

/* Boundary Condition */

for( i=0 ; i<idvs+1  ; i++ ){
    s1[i] = 1.0;
    s3[i] = 1.0;
}
for( j=0 ; j<jdvs+1  ; j++ ){
    s2[j] = 1.0;
    s4[j] = 1.0;
}

ierr = ASL_dopdh2(dlmd, f, idvs, jdvs, x0, x1, y0, y1, s1, s2, s3, s4, imax, eps, u, imx, isw, iwk, wk);

nxi = 5;
nyi = 6;
printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );

printf( "\n\tSolution u[i]\n\n" );

for( j=jdvs ; j>=0 ; j -= nyi )
{
    printf( "\t%6d", j );
    for( i=0 ; i<idvs+1  ; i += nxi )
        printf( "%8.3g", u[i + (imx+1)*j] );
    printf( "\n" );
}
printf( "    y-Direction      0       5      10" );
printf( "      15      20\n" );
```

```
        printf( "\t     x-Direction\n" );
        printf( "\n" );

        free(s1 );
        free(s2 );
        free(s3 );
        free(s4 );
        free(u  );
        free(isw);
        free(iwk);
        free(wk );

        return 0;
    }
```

(d) Output results

```
        *** ASL_dopdh2 ***

         ** Input **

        isw   =      1      0      0      0
        dlmd  =        1
        nx,ny =       20       30
        x0,x1 =        0      1.2
        y0,y1 =        0      1.8
        imax  =   1000
        eps   =    1e-09

         ** Output **

        ierr =      0

        Solution u[i]

            30   0.931   0.742   0.668   0.693     0.9
            24   0.754  -0.128    -0.5  -0.313    0.67
            18   0.731  -0.313  -0.777  -0.524   0.639
            12   0.763  -0.201  -0.656  -0.417    0.67
             6   0.812-0.00269  -0.417   -0.22   0.718
             0   0.844   0.121   -0.27 -0.0966   0.751
        y-Direction      0       5      10      15      20
             x-Direction
```

2.5.2 ASL_dopdh3, ASL_ropdh3
Three-Dimensional Inhomogeneous Helmholtz Equation

(1) **Function**

This function uses a three-dimensional seven-point difference in a given rectangular area to solve the following inhomogeneous Helmholtz equation.

$$u_{xx} + u_{yy} + u_{zz} + \lambda u = f(x, y, z)$$

(2) **Usage**

Double precision:

ierr = ASL_dopdh3 (dlmd, func, nx, ny, nz, xl, xu, yl, yu, zl, zu, s1, s2, s3, s4, s5, s6, imx, jmx, imax, eps, u, isw, iw, w);

Single precision:

ierr = ASL_ropdh3 (dlmd, func, nx, ny, nz, xl, xu, yl, yu, zl, zu, s1, s2, s3, s4, s5, s6, imx, jmx, imax, eps, u, isw, iw, w);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	dlmd	{ D / R }	1	Input	Value of the coefficient of u in the difference equation λ.
2	func	–	–	Input	Name of the function f(x,y,z) that defines the differential equations as a function of x, y and z.
3	nx	I	1	Input	Number of division in X-direction n_x
4	ny	I	1	Input	Number of division in Y-direction n_y
5	nz	I	1	Input	Number of division in Z-direction n_z
6	xl	I	1	Input	The value of X along the lower point of the domain.
7	xu	I	1	Input	The value of X along the upper upper of the domain.
8	yl	I	1	Input	The value of Y along the lower point of the domain.
9	yu	I	1	Input	The value of Y along the upper point of the domain.
10	zl	I	1	Input	The value of z along the lower point of the domain.
11	zu	I	1	Input	The value of Y along the upper point of the domain.
12	s1	I*	See Contents	Input	Value of Dirichlet boundary condition of bottom surface (when isw[0] $\neq$ 1)When isw[0] = 1, the area is only allocated. **Size**: (imx + 1) $\times$ (nz + 1)

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
13	s2	I*	See Contents	Input	Value of Dirichlet boundary condition of right surface (when isw[1] $\neq$ 1)When isw[1] = 1, the area is only allocated. **Size**: (jmx + 1) × (nz + 1)
14	s3	I*	See Contents	Input	Value of Dirichlet boundary condition of top surface (when isw[2] $\neq$ 1)When isw[2] = 1, the area is only allocated. **Size**: (imx + 1) × (nz + 1)
15	s4	I*	See Contents	Input	Value of Dirichlet boundary condition of left surface (when isw[3] $\neq$ 1)When isw[3] = 1, the area is only allocated. **Size**: (jmx + 1) × (nz + 1)
16	s5	I*	See Contents	Input	Value of Dirichlet boundary condition of front surface (when isw[4] $\neq$ 1)When isw[4] = 1, the area is only allocated. **Size**: (imx + 1) × (ny + 1)
17	s6	I*	See Contents	Input	Value of Dirichlet boundary condition of back surface (when isw[5] $\neq$ 1)When isw[5] = 1, the area is only allocated. **Size**: (imx + 1) × (ny + 1)
18	imx	I	1	Input	Adjustable dimension of array s1, s3, s5, s6
19	jmx	I	1	Input	Adjustable dimension of array s2, s4
20	imax	I	1	Input	Maximum number of iterations when using an iterative method to solve simultaneous linear equations
21	eps	I	1	Input	Truncation residual norm $\| \boldsymbol{b} - A\boldsymbol{u} \| / \| \boldsymbol{b} \|$ **Specifiable range:** ≥Underflow decision value (See Note (e))
22	u	$\left\{\begin{matrix}\text{D*}\\ \text{R*}\end{matrix}\right\}$	See Contents	Output	Solution vector$u(x_i, y_j, z_k)$ u$(i, j, k) = u\ (x_L + i(x_U - x_L)/n_x\ ,\ y_L + j(y_U - y_L)/n_y\ ,\ z_L + k(z_U - z_L)/n_z)$ **Size**: (imx + 1) × (jmx + 1) × (nz + 1)

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
23	isw	I*	6	Input	Boundary condition selection switch isw[0] =0 : Add Dirichlet condition to bottom surface. =1 : Add Neumann condition to bottom surface. isw[1] =0 : Add Dirichlet condition to right surface. =1 : Add Neumann condition to right surface. isw[2] =0 : Add Dirichlet condition to top surface. =1 : Add Neumann condition to top surface. isw[3] =0 : Add Dirichlet condition to left surface. =1 : Add Neumann condition to left surface. isw[4] =0 : Add Dirichlet condition to front surface. =1 : Add Neumann condition to front surface. isw[5] =0 : Add Dirichlet condition to back surface. =1 : Add Neumann condition to back surface.
24	iw	I*	7	Work	Work area
25	w	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $15 \times (\text{nx}+1) \times (\text{ny}+1) \times (\text{nz}+1) + 5$
26	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{nx} \geq 2$, $\text{ny} \geq 2$, $\text{nz} \geq 2$

(b) For the boundary conditions to be added to the edges, the Dirichlet condition must be added to at least one edge.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1400	A diagonal element produced by ILU decomposition became less than ϵ times the absolute value of the original diagonal element. (See Note (c))	Replaces diagonal element produced by ILU decomposition with (original diagonal element), $\times\epsilon$ and processing continues.
2000	Maximum number of iterations has been reached.	Processing continues without performing acceleration.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
4000	The norm of right-hand side vector is greater than $\sqrt{\text{overflow decision value}}$. (See Notes (d), (e))	
4100	The norm of the residual is greater than $\sqrt{\text{overflow decision value}}$. (See Notes (d), (e))	
4200	$\parallel (\boldsymbol{r}, \boldsymbol{r^*}) \parallel$ is smaller than underflow decision value.	
4210	$\parallel (\boldsymbol{r}, \boldsymbol{r^*}) \parallel$ is greater than overflow decision value.	
4300	$\parallel (A\boldsymbol{p}, \boldsymbol{p^*}) \parallel$ is smaller than underflow decision value.	
4310	$\parallel (A\boldsymbol{p}, \boldsymbol{p^*}) \parallel$ is greater than overflow decision value.	
5000	Any of diagonal element has absolute value smaller than underflow decision value.	

(6) **Notes**

(a) The actual name of function f(x, y), that defines the differential equations, must be declared in the user program, and the actual function must be created. For the high-order simultaneous ordinary differential equations:

$$u_{xx} + u_{yy} + u_{zz} + \lambda u = f(x, y, z)$$

this function f(x, y, z) (in double-precision) should be created as follows:

```
double FORTRAN func(double *x, double *y, double *z)
{
    return f(x,y,z);
}
```

where the following correspondences are assumed.
$x \leftrightarrow$ *x,$y \leftrightarrow$ *y, $z \leftrightarrow$ *z

Example:

$$f(x, y) = 6x + 6y + 6z$$

Therefore, define the function as follows:

```
double FORTRAN func(double *x, double *y, double *z)
{
    return 6.0*(*x) + 6.0*(*y) + 6.0*(*z);
}
```

(b) For the Dirichlet conditions s1, s2, s3, s4, s5, and s6, $(nx+1)\times(nz+1)$ (s1, s3) values and $(ny+1)\times(nz+1)$(s2, s4) values and $(nx+1)\times(ny+1)$(s5, s6) value, excluding values at the vertices of the virtual grid, are stored in arrays sequentially in ascending order of coordinate values.

(c) The singularity prevention constant ϵ is set to 0.1. (See the high-speed function version (4.1.2.2).)

(d) The l^2 norm shown below is used for the norm.

$$\|\boldsymbol{r}\|_2 = \left(\sum_{i=1}^{n} {r_i}^2\right)^{1/2} \quad where, \boldsymbol{r} = (r_1, r_2, r_3, \cdots, r_n)^T$$

(e) (Maximum value$\times 10^{-3}$) is set for the overflow decision value and (Minimum positive value$\times 10^{3}$) is set for the underflow decision value.

(7) **Example**

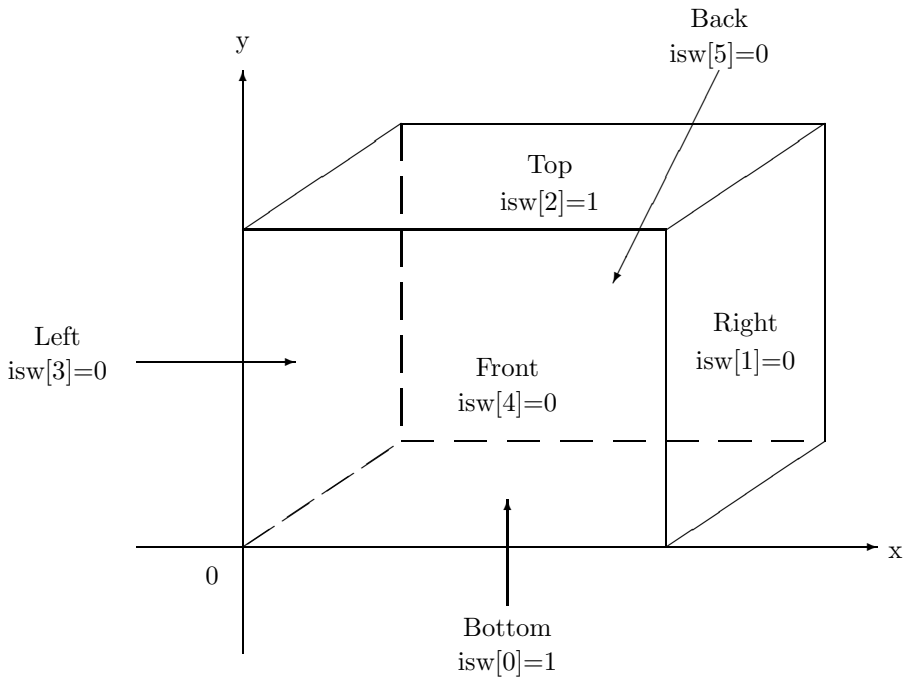
(a) Problem (three-dimensional Poisson equation)
Solve the following three-dimensional Poisson equation :

$$u_{xx} + u_{yy} + u_{zz} = f(x, y, z)$$

under the boundary conditions below.

$$\begin{cases} \frac{\partial u}{\partial y} = 0 & \text{(on the bottom and top surface)} \\ u = 1 & \text{(on the other surfaces)} \end{cases}$$

The given three-dimensional area is discretized by an nx × ny × nz orthogonal grid as shown in the following figure.



(b) Input data

dlmd=0.0,

nx=20, ny=30, nz=40,

xl=0.0, xu=0.8,

yl=0.0, yu=0.6,

zl=0.0, zu=1.2,

imax=200, eps=1.e-10 and

isw=(1, 0, 1, 0, 0, 0).

(c) Main program

```
/*      C interface example for ASL_dopdh3 */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x, double *y, double *z)
#else
double f(x,y,z)
double *x;
double *y;
double *z;
#endif
{
    return 6.0e0*(*x) + 6.0e0*(*y) + 6.0e0*(*z);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double dlmd;
    int idvs;
    int jdvs;
    int kdvs;
    double x0;
    double x1;
    double y0;
    double y1;
    double z0;
    double z1;
    double *s1;
    double *s2;
    double *s3;
    double *s4;
    double *s5;
    double *s6;
    int imax;
    double eps;
    double *u;
    int imx=50;
    int jmx=50;
    int *isw;
    int *iwk;
    double *wk;
    int ierr;

    int i,j,k,nxi,nyi,nzi,kst,jst;
    FILE *fp;

    fp = fopen( "dopdh3.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dopdh3 ***\n" );
    printf( "\n     ** Input **\n\n" );

    isw = ( int * )malloc((size_t)( sizeof(int) * 6 ));
    if( isw == NULL )
    {
        printf( "no enough memory for array isw\n" );
        return -1;
    }
```

```
fscanf( fp, "%d", &isw[0]);
fscanf( fp, "%d", &isw[1]);
fscanf( fp, "%d", &isw[2]);
fscanf( fp, "%d", &isw[3]);
fscanf( fp, "%d", &isw[4]);
fscanf( fp, "%d", &isw[5]);

fscanf( fp, "%lf",    &dlmd                );
fscanf( fp, "%d,%d,%d",  &idvs,&jdvs,&kdvs );
fscanf( fp, "%lf,%lf",&x0,&x1              );
fscanf( fp, "%lf,%lf",&y0,&y1              );
fscanf( fp, "%lf,%lf",&z0,&z1              );
fscanf( fp, "%d",     &imax                );
fscanf( fp, "%lf",    &eps                 );

printf( "\tisw   = %6d %6d %6d %6d %6d %6d\n",
    isw[0],isw[1],isw[2],isw[3],isw[4],isw[5] );
printf( "\tdlmd  = %8.3g\n",      dlmd               );
printf( "\tidvs  =   %6d\n",idvs                     );
printf( "\tjdvs  =   %6d\n",jdvs                     );
printf( "\tkdvs  =   %6d\n",kdvs                     );
printf( "\tx0,x1 = %8.3g %8.3g\n",x0,x1              );
printf( "\ty0,y1 = %8.3g %8.3g\n",y0,y1              );
printf( "\tz0,z1 = %8.3g %8.3g\n",z0,z1              );
printf( "\timax  = %6d\n",        imax               );
printf( "\teps   = %8.3g\n",      eps                );

s1 = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(kdvs+1) ));
if( s1 == NULL )
{
    printf( "no enough memory for array s1\n" );
    return -1;
}

s2 = ( double * )malloc((size_t)( sizeof(double) * (jmx+1)*(kdvs+1) ));
if( s2 == NULL )
{
    printf( "no enough memory for array s2\n" );
    return -1;
}

s3 = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(kdvs+1) ));
if( s3 == NULL )
{
    printf( "no enough memory for array s3\n" );
    return -1;
}

s4 = ( double * )malloc((size_t)( sizeof(double) * (jmx+1)*(kdvs+1) ));
if( s4 == NULL )
{
    printf( "no enough memory for array s4\n" );
    return -1;
}

s5 = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(jdvs+1) ));
if( s5 == NULL )
{
    printf( "no enough memory for array s3\n" );
    return -1;
}

s6 = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(jdvs+1) ));
if( s6 == NULL )
{
    printf( "no enough memory for array s4\n" );
    return -1;
}

u = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(jmx+1)*(kdvs+1) ));
if( u == NULL )
{
    printf( "no enough memory for array u\n" );
    return -1;
}

iwk = ( int * )malloc((size_t)( sizeof(int) * 7 ));
if( iwk == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

wk  = ( double * )malloc((size_t)( sizeof(double) * (imx+1)*(jmx+1)*(kdvs+1)*13+5 ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}
```

```
    fclose( fp );

    /* Boundary Condition */

    for( i=0 ; i<idvs+1  ; i++ ){
        for( j=0 ; j<jdvs+1  ; j++ ){
            s5[i+(imx+1)*j] = 1.0e0;
            s6[i+(imx+1)*j] = 1.0e0;
        }
    }
    for( j=0 ; j<jdvs+1  ; j++ ){
        for( k=0 ; k<kdvs+1  ; k++ ){
            s2[j+(jmx+1)*k] = 1.0e0;
            s4[j+(jmx+1)*k] = 1.0e0;
        }
    }
    for( i=0 ; i<idvs+1  ; i++ ){
        for( k=0 ; k<kdvs+1  ; k++ ){
            s1[i+(imx+1)*k] = 1.0e0;
            s3[i+(imx+1)*k] = 1.0e0;
        }
    }

    ierr = ASL_dopdh3
    (dlmd,f,idvs,jdvs,kdvs,x0,x1,y0,y1,z0,z1,s1,s2,s3,s4,s5,s6,
     imx,jmx,imax,eps,u,isw,iwk,wk);

    nxi = 5;
    nyi = 6;
    nzi = 5;
    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );

    printf( "\n\tSolution u[i]\n\n" );

    for( k=0 ; k<kdvs+1 ; k += nzi )
    {
        kst = k*(imx+1)*(jmx+1);
        printf( "\n\tz-Direction = %6d\n\n",k );
        for( j=jdvs ; j>=0 ; j -= nyi )
        {
            printf( "\t%6d", j );
            jst = (imx+1)*j;
            for( i=0 ; i<idvs+1  ; i += nxi )
            {
                printf( "%8.3g", u[kst + jst + i] );
            }
        printf( "\n" );
        }
        printf( "\n" );
        printf( "    y-Direction      0       5      10" );
        printf( "      15      20\n" );
        printf( "\t     x-Direction\n" );
        printf( "\n" );
    }

    free(s1 );
    free(s2 );
    free(s3 );
    free(s4 );
    free(s5 );
    free(s6 );
    free(u  );
    free(isw);
    free(iwk);
    free(wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dopdh3 ***

 ** Input **

isw   =      1      0      1      0      0      0
dlmd  =        0
idvs  =       20
jdvs  =       30
kdvs  =       20
x0,x1 =        0      0.8
y0,y1 =        0      0.6
z0,z1 =        0      1.2
imax  =    200
eps   =    1e-05

 ** Output **

ierr =      0
```

```
Solution u[i]


z-Direction =      0

    30    0.98   0.916   0.887   0.895   0.968
    24   0.981   0.919   0.891   0.899   0.969
    18   0.983   0.925   0.897   0.905   0.971
    12   0.986   0.932   0.905   0.911   0.974
     6   0.988   0.938   0.912   0.917   0.976
     0   0.989   0.941   0.915   0.921   0.977

y-Direction      0       5      10      15      20
     x-Direction


z-Direction =      5

    30   0.922   0.639   0.519   0.583   0.897
    24   0.925   0.647   0.529   0.591   0.899
    18    0.93   0.665   0.549   0.609   0.904
    12   0.935   0.685   0.572   0.628    0.91
     6    0.94   0.702   0.593   0.646   0.914
     0   0.942   0.711   0.602   0.654   0.917

y-Direction      0       5      10      15      20
     x-Direction


z-Direction =     10

    30   0.893   0.504   0.347   0.444   0.866
    24   0.895   0.513   0.357   0.453   0.869
    18     0.9   0.532   0.379   0.471   0.874
    12   0.906   0.553   0.405   0.493    0.88
     6   0.911   0.572   0.427   0.512   0.885
     0   0.914   0.581   0.438   0.521   0.887

y-Direction      0       5      10      15      20
     x-Direction


z-Direction =     15

    30    0.89   0.511   0.366   0.454   0.864
    24   0.892   0.519   0.375   0.463   0.867
    18   0.897   0.536   0.395    0.48   0.872
    12   0.903   0.556   0.419     0.5   0.877
     6   0.907   0.574   0.439   0.517   0.882
     0    0.91   0.582   0.449   0.526   0.885

y-Direction      0       5      10      15      20
     x-Direction


z-Direction =     20

    30   0.958    0.85   0.812   0.829   0.947
    24    0.96   0.853   0.816   0.833   0.948
    18   0.962   0.859   0.823   0.839    0.95
    12   0.964   0.866    0.83   0.845   0.952
     6   0.966   0.872   0.837   0.851   0.954
     0   0.967   0.875   0.841   0.855   0.956

y-Direction      0       5      10      15      20
     x-Direction
```

Chapter 3

NUMERICAL DIFFERENTIALS

3.1 INTRODUCTION

This chapter describes functions that obtain numerical differentials of functions at given points.
This library provides the following function for obtaining numerical differentials of a function of one variable.

(1) Numerical differentials of a function
This function obtains arbitrary order differential values of a function of one variable at a given point.
This library provides the following functions for obtaining numerical differentials of a single function of many variables.

(2) Gradient vector of a function of many variables

(3) Hessian matrix of a function of many variables
The function in (2) obtains the first order partial differential values (gradient vector) of a function of many variables at given points.
The function in (3) obtains the second order partial differential values (Hessian matrix) of a function of many variables at given points.
This library provides the following function for obtaining numerical differentials of multiple functions of many variables.

(4) Jacobian matrix of multiple functions of many variables
This function obtains the first order partial differential values (Jacobian matrix) of multiple functions of many variables at given points.

3.1.1 Notes

(1) A value that is larger than the default value should be entered as the required relative precision.

(2) The functions, whose name passed to function, are created as follows.
Example:

Numerical differentials of a function (Let f has the same name in the main program and the function)

- Main program

```
≀
ierr = { ASL_dqfodx
       { ASL_rqfodx } (f, ···);
≀
```

- Function

```
double FORTRAN f (double x)
{
  ≀
  return f(*x);
}
```

Example:

Gradient vector of a function of many variables and Hessian matrix of a function of many variables (Let f has the same name in the main program and the function)

- Main program

```
≀
ierr = { ASL_dqmo**
       { ASL_rqmo** } (f, ···);
≀
```

- Function

```
double FORTRAN f (double x)
{
  ≀
  return f(x[0], ···, x[nx − 1]);
}
```

Example:

Jacobian matrix of multiple functions of many functions of many variables (Let sub has the same name in the main program and the function)

- Main program

  ≀

  $$\text{ierr} = \left\{\begin{matrix}\text{ASL\_dqmojx}\\ \text{ASL\_rqmojx}\end{matrix}\right\}(\boxed{\text{sub}}, \cdots);$$

  ≀

- Function

  void FORTRAN $\boxed{\text{sub}}$(double $*$x, int $*$nx, double $*$f, int $*$nf)

  {

  ≀

  f[0] $= f_1$(x[0], $\cdots$, x[($*$nx) $-$ 1]);

  ≀

  f[($*$nf) $-$ 1] $= f_{(*\text{nf})}$(x[0], $\cdots$, x[($*$nx) $-$ 1]);

  }

3.1.2 Algorithms Used

3.1.2.1 Richardson's extrapolation

Obtain the differential value $f^{(1)}(x)$ of a function $f(x)$ at x by using differences.
From the Taylor expansion of $f(x+h)$ and $f(x-h)$, $f^{(1)}(x)$ can be written as:

$$f^{(1)}(x) = \frac{f(x+h) - f(x-h)}{2h} + c_1 h^2 + c_2 h^4 + \cdots + c_i h^{2i} + \cdots \tag{3.1}$$

where:

$$c_i = -\frac{f^{(2i+1)}(x)}{(2i+1)!} \ (i = 1, 2, \cdots)$$

Let the central difference of the step size h be expressed as:

$$D_0^{(0)} = \frac{f(x+h) - f(x-h)}{2h}$$

Then, the error of $D_0^{(0)}$ and $f^{(1)}(x)$ is on the order of h^2 from equation (3.1).
Now, divide the step size in half by setting it to $\frac{h}{2}$ and let the central difference be expressed as:

$$D_0^{(1)} = \frac{f(x+\frac{h}{2}) - f(x-\frac{h}{2})}{2(\frac{h}{2})}$$

If $D_0^{(0)}$ and $D_0^{(1)}$ are used to define the quantity:

$$D_1^{(0)} = \frac{1}{3}(4D_0^{(1)} - D_0^{(0)})$$

then, from equation (3.1) and the equation obtained by replacing h by $\frac{h}{2}$ in equation (3.1), express the error of this $D_1^{(0)}$ and $f^{(1)}(x)$ as:

$$f^{(1)}(x) - D_1^{(0)} = -\frac{1}{4}c_2 h^4 + \cdots$$

which is on the order of h^4. $D_1^{(0)}$ gives a higher order approximation to $f^{(1)}(x)$ than $D_0^{(0)}$.
Generalizing this procedure, by sequentially eliminating the high order terms of h in equation (3.1), a high order approximation of $f^{(1)}(x)$ is sequentially obtained. First, start with a suitable h and calculate:

$$D_0^{(k)} = \frac{f(x+\frac{h}{2^k}) - f(x-\frac{h}{2^k})}{2(\frac{h}{2^k})} \ (k = 0, 1, 2, \cdots)$$

Next, for $m = 1, 2, \cdots$ calculate the following (Richardson extrapolation):

$$\begin{aligned} D_m^{(k)} &= \frac{4^m D_{m-1}^{(k+1)} - D_{m-1}^{(k)}}{4^m - 1} \\ &= D_{m-1}^{(k+1)} + \frac{1}{4^m - 1}(D_{m-1}^{(k+1)} - D_{m-1}^{(k)}) \end{aligned}$$

(See Figure 3−1) The error of $D_m^{(0)}$ is on the order of $h^{2(m+1)}$.

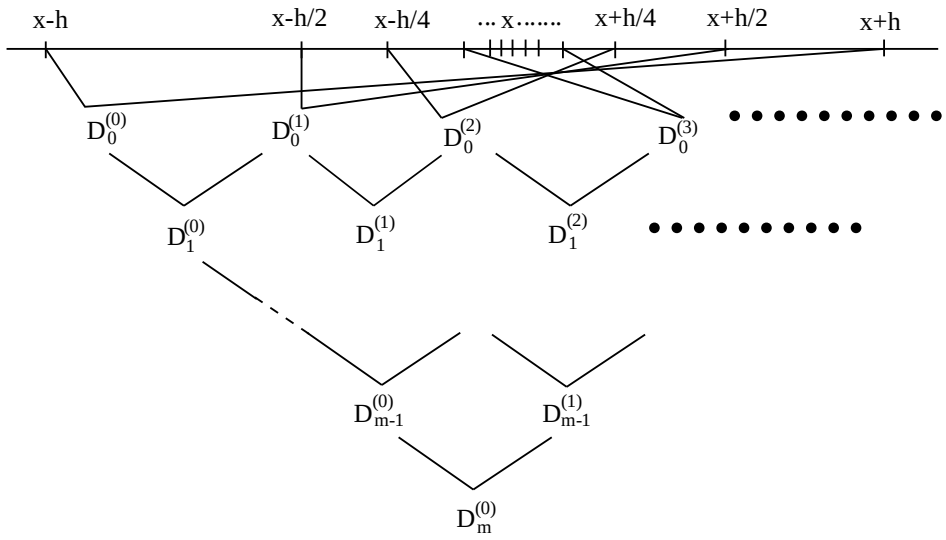


Figure 3−1 Richardson's Extrapolation

Use the following condition for determining convergence:

$$\frac{|D_m^{(0)} - D_{m-1}^{(0)}|}{|D_m^{(0)}|} \leq 3 \times \text{EPSN} \quad \text{(where EPSN is the required relative precision)}$$

Assume that the sequence has converged when this condition holds and use $D_m^{(0)}$ as the approximation of $f^{(1)}(x)$.

3.1.2.2 Numerical differentials of a function

Consider a high order differential formula for obtaining a differential of an arbitrary order.
If n is an even number, let the n-th order differential formula be as follows:

$$f^{(n)}(x) \sim \frac{\sum_k b_k(f(x+kh) + f(x-kh))}{h^n}$$

where, k satisfies the relationship $1 \leq k \leq \frac{n}{2}$.
From the binomial theorem, the coefficients b_k are considered to be:

$$\begin{aligned} b_k &= {}_{n/2}C_k \quad (k \text{ is an odd number}) \\ b_k &= -{}_{n/2}C_k \quad (k \text{ is an even number}) \end{aligned}$$

If n is an odd number, let the n-th order differential formula be as follows:

$$f^{(n)}(x) \sim \frac{\sum_k b_k(f(x+kh) - f(x-kh))}{2h^n}$$

where, k satisfies the relationship $1 \leq k \leq \lfloor \frac{n}{2} \rfloor + 1$ (where the notation $\lfloor x \rfloor$ represents the maximum integer that does not exceed x).
From the binomial theorem, the coefficients b_k are considered to be:

$$\begin{aligned} b_k &= {}_{\lfloor \frac{n}{2} \rfloor + 1}C_k \quad (k \text{ is an odd number}) \\ b_k &= -{}_{\lfloor \frac{n}{2} \rfloor + 1}C_k \quad (k \text{ is an even number}) \end{aligned}$$

and then when $n \geq 5$, the following values are taken by using the b_k values that had been obtained:

$$b_k \leftarrow b_k - b_{k-2} \ (k = \lfloor \frac{n}{2} \rfloor + 1, \cdots, 3)$$

Richardson's extrapolation is performed for the n-th order differential formula obtained in this way.
In addition, processing is aborted if the step size becomes smaller than the following value:

$\max(x\varepsilon, \varepsilon^{\frac{1}{n+1}})$ (ε is the unit for determining error)

3.1.2.3 Gradient vector of a function of many variables

For multiple variables $x_i, \cdots, x_{nx}$, fix the variables other than x_i $(i = 1, \cdots, nx)$. Then, obtain the first order differential for x_i by using Richardson's extrapolation. That is, the gradient vector for function f in terms of x_i is given by:

$$\frac{\partial f}{\partial x_i} \ (i = 1, \cdots, nx)$$

Processing is aborted if the step size becomes smaller than the following value when Richardson's extrapolation is performed:

$$\max_{i=1,\cdots,nx}(x_i\varepsilon, \varepsilon^{0.25}) \ (\varepsilon \text{ is the unit for determining error})$$

3.1.2.4 Hessian matrix of a function of multiple variables

Obtain the first order differential values for each of the variables $x_1, \cdots, x_{nx}$. Perform Richardson's extrapolation for that first order differential value.
For multiple variables $x_1, \cdots, x_{nx}$, fix the variables other than x_i $(i = 1, \cdots, nx)$ and obtain the first order differential in terms of x_i $(i = 1, \cdots, nx)$ for the first order differential values of each of the variables.
That is, the (i, j)-th element of the Hessian matrix for the function is given by:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \ (i = 1, \cdots, nx; \ j = 1, \cdots, nx)$$

Processing is aborted if the step size becomes smaller than the following value when Richardson's extrapolation is performed:

$$\max_{i=1,\cdots,nx}(x_i\varepsilon, \varepsilon^{0.25}) \ (\varepsilon \text{ is the unit for determining error})$$

3.1.2.5 Jacobian matrix of a function of multiple variables

Using an algorithm similar to the one described for obtaining the gradient vector, obtain the first order differential values of each of the multiple functions $f_1, \cdots, f_{nf}$ in terms of each of the variables $x_1, \cdots, x_{nx}$.
That is, the (i, j)-th element of the Jacobian matrix is given by:

$$\frac{\partial f_i}{\partial x_j} \ (i = 1, \cdots, nf; \ j = 1, \cdots, nx)$$

Processing is aborted if the step size becomes smaller than the following value when Richardson's extrapolation is performed:

$$\max_{i=1,\cdots,nx}(x_i\varepsilon, \varepsilon^{0.25}) \ (\varepsilon \text{ is the unit for determining error})$$

3.1.3 Reference Bibliography

(1) Hitotsumatsu Shin and Togawa Hayato editors, “Error in Numerical Calculations”, Kyoritsu Shuppan, (1975).

(2) Mori Masatake, “Numerical Calculation Programming”, Iwanami Shoten, (1986).

3.2 NUMERICAL DIFFERENTIALS

3.2.1 ASL_dqfodx, ASL_rqfodx
Numerical Differentials of a Function

(1) **Function**

ASL_dqfodx or ASL_rqfodx uses differences and Richardson's extrapolation to obtain the n-th order differential value $f^{(n)}(x)$ of the function $f(x)$.

(2) **Usage**

Double precision:

```
ierr = ASL_dqfodx  (f, x, n, eps, &mr, h, &del, wk);
```

Single precision:

```
ierr = ASL_rqfodx  (f, x, n, eps, &mr, h, &del, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name of function f(x) that defines the function $f(x)$ of x. (See Notes (a))
2	x	{D / R}	1	Input	Differentiation point x.
3	n	I	1	Input	Differential order n
4	eps	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64)
5	mr	I*	1	Input	Maximum number of iterations (Default value: 100)
				Output	Actual number of iterations
6	h	{D / R}	1	Input	Initial step size
7	del	{D* / R*}	1	Output	The n-th order differential value $f^{(n)}(x)$ of the function at the differentiation point
8	wk	{D* / R*}	See Contents	Work	Work area (See Notes (b)) **Size**: $\lfloor n/2 \rfloor + mr + 2$ ($\lfloor x \rfloor$ represents the maximum integer that does not exceed x.)
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) eps $\geq \varepsilon \times 64$
(except when 0.0 is entered in order to set eps to the default value,
ε is the unit for determining error.)

(b) mr > 0
(except when 0.0 is entered in order to set mr to the default value)

(c) n > 0

(d) h $> \max(\mathrm{x} \times \varepsilon, \varepsilon^{1/(\mathrm{n}+3)})$
(ε is the unit for determining error.)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) or (b) was not satisfied.	Processing is performed with the corresponding default value set.
2000	The error became large before the required precision was achieved or the step size was too small.	The calculation value at that time is output, and processing is aborted.
2500	The maximum number of iterations was reached without the required precision being achieved.	
3000	Restriction (c) or (d) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) The function f should be created as follows.

```
double FORTRAN f(double *x)
{
    return(f(*x));
}
```

(b) When reserving the area for the eighth argument wk as an array, let the size of the array be $\lfloor \mathrm{n}/2 \rfloor + 102$ if mr ≤ 0.

(c) Enter a value larger than the default value as the required relative precision.

(d) The value of the sixth argument h should be on the order of 0.5 to 1.0.

(e) For the convergence decision, consider that the sequence has converged when the following relationship holds:
{| Extrapolation value | − | Previous extrapolation value |≤| eps × Extrapolation value |}

(f) If a default value is shown in the Contents column of the table describing the arguments, the default value will be set if 0 is entered for an integer type argument or 0.0 is entered for a real type argument.

(g) Since precision often decreases if n > 5 is set for the differential order when the single precision function is used, the double precision function should be used. However, n ≤ 15 should be set even when the double precision function is used.

(7) **Example**

(a) Problem

Obtain an approximate value of the fifth order differential of the function $f(x) = \sin(x)$ at $x = -3.75$.

(b) Input data

Function name: f.

x=−3.75, n=5, eps=0.0, mr=15 and h=0.5.

(c) Main program

```
/*      C interface example for ASL_dqfodx */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return sin(*x);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double x;
    int n;
    double epsn;
    int mr;
    double h;
    double d;
    double *wk;
    int ierr;
    int nwk;
    FILE *fp;

    fp = fopen( "dqfodx.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dqfodx ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=5;

    fscanf( fp, "%lf", &x );
    fscanf( fp, "%lf", &epsn );
    fscanf( fp, "%d", &mr );
    fscanf( fp, "%lf", &h );
    printf( "\tOrder of Differentiation = %6d\n",n );
    printf( "\tx   = %8.3g\n",x );
    printf( "\teps = %8.3g\n",epsn );
    printf( "\tmr  =   %6d\n",mr );
    printf( "\th   = %8.3g\n",h );

    nwk=n/2+mr+2;
    wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    fclose( fp );

    ierr = ASL_dqfodx(f, x, n, epsn, &mr, h, &d, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );

    printf( "\n\tIteration Number   =   %6d\n",mr );
    printf( "\tDifferential Value = %8.3g\n",d );
```

```
    free( wk  );

    return 0;
}
```

(d) Output results

```
*** ASL_dqfodx ***

 ** Input **

Order of Differentiation =      5
x   =    -3.75
eps =        0
mr  =       15
h   =      0.5

 ** Output **

ierr =      0

Iteration Number   =         3
Differential Value =    -0.821
```

3.2.2 ASL_dqmogx, ASL_rqmogx
Gradient Vector of a Function of Many Variables

(1) **Function**

ASL_dqmogx or ASL_rqmogx uses differences and Richardson's extrapolation to obtain the gradient vector $\partial f/\partial x_j$ of a function of many variables, $f(\boldsymbol{x})(\boldsymbol{x} = \{x_j\}; j = 1, \cdots, nx)$.

(2) **Usage**

Double precision:

ierr = ASL_dqmogx (f, x, nx, eps, mr, h, grad, wk);

Single precision:

ierr = ASL_rqmogx (f, x, nx, eps, mr, h, grad, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name of function f(x) that defines the function $f(\boldsymbol{x})$ of $\boldsymbol{x}$. (See Notes (a))
2	x	{D∗ / R∗}	nx	Input	Differentiation point $x(x_1, \cdots, x_{nx})$.
3	nx	I	1	Input	Number nx of independent variables x
4	eps	{D / R}	1	Input	Required relative precision (Default value: (Unit for determining error)$^{\frac{2}{3}}$
5	mr	I	1	Input	Maximum number of iterations (Default value: 100)
6	h	{D / R}	1	Input	Initial step size
7	grad	{D∗ / R∗}	nx	Output	Gradient vector $\partial f/\partial \boldsymbol{x}$ of the function at the differentiation point
8	wk	{D∗ / R∗}	mr + 1	Work	Work area (See Notes (b))
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) eps $\geq \varepsilon \times 64$
(except when 0.0 is entered in order to set eps to the default value,
ε is the unit for determining error.)

(b) mr > 0
(except when 0.0 is entered in order to set mr to the default value)

(c) nx > 0

(d) h $\geq \max(x_i \times \varepsilon, \varepsilon^{0.25})$ $(i = 1, \cdots, \mathrm{nx})$
(ε is the unit for determining error.)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) or (b) was not satisfied.	Processing is performed with the corresponding default value set.
2000	The error became large before the required precision was achieved or the step size was too small.	The calculation value at that time is output, and processing is aborted.
2500	The maximum number of iterations was reached without the required precision being achieved.	
3000	Restriction (c) or (d) was not satisfied.	

(6) **Notes**

(a) The function f should be created as follows.

```
double FORTRAN f(double *x)
{
    return(f[0],..., f[nx-1]);
}
```

(b) When reserving the area for the eighth argument wk as an array, let the size of the array be 101 if mr ≤ 0.

(c) Enter a value larger than the default value as the required relative precision.

(d) The value of the sixth argument h should be on the order of 0.5 to 1.0.

(e) For the convergence decision, consider that the sequence has converged when the following relationship holds:
{| Extrapolation value | − | Previous extrapolation value |≤| eps × Extrapolation value |}

(f) If a default value is shown in the Contents column of the table describing the arguments, the default value will be set if 0 is entered for an integer type argument or 0.0 is entered for a real type argument.

(g) If several different errors occur at the same time, the most severe error value will be output for the error indicator, and the other error information may disappear.

(7) **Example**

(a) Problem

Obtain gradient vector of the function $f(x_1, x_2, x_3) = x_1x_2x_3$ at $(x_1, x_2, x_3) = (2.1, 8.59, 0.315)$.

(b) Input data

Function name: f.

x[0]=2.1, x[1]=8.59, x[2]=0.315, nx=3, eps=1.0e−9, mr=15 and h=0.5.

(c) Main program

```
/*      C interface example for ASL_dqmogx */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return (x[0]*x[1]*x[2]);
}
#ifdef __cplusplus
}
#endif


int main()
{
    double *x1;
    int nx;
    double epsn;
    int mr;
    double h;
    double *grad;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dqmogx.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dqmogx ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=3;
    mr=15;
    x1 = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x1  == NULL )
    {
        printf( "no enough memory for array x1\n" );
        return -1;
    }
    grad  = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( grad  == NULL )
    {
        printf( "no enough memory for array grad\n" );
        return -1;
    }
    wk  = ( double * )malloc((size_t)( sizeof(double) * (mr+1) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x1[i] );
    }
    fscanf( fp, "%lf", &epsn );
    fscanf( fp, "%lf", &h );
    printf( "\tn   =   %6d\n",nx );
    printf( "\tx   = " );
    for( i=0 ; i<nx ; i++ )
```

```
    {
        printf( "%8.3g",x1[i] );
    }
    printf( "\n" );
    printf( "\teps = %8.3g\n",epsn );
    printf( "\tmr  =   %6d\n",mr );
    printf( "\th   = %8.3g\n",h );

    fclose( fp );

    ierr = ASL_dqmogx(f, x1, nx, epsn, mr, h, grad, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );

    printf( "\n" );
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\tgrad[%6d] = %8.3g\n", i,grad[i] );
    }

    free( x1 );
    free( grad );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dqmogx ***

 ** Input **

n   =        3
x   =      2.1    8.59   0.315
eps =    1e-09
mr  =       15
h   =      0.5

 ** Output **

ierr =      0

grad[     0] =     2.71
grad[     1] =    0.662
grad[     2] =       18
```

3.2.3 ASL_dqmohx, ASL_rqmohx
Hessian of a Function of Many Variables

(1) **Function**

ASL_dqmohx or ASL_rqmohx uses Richardson's extrapolation to obtain the Hessian $\partial^2 f/(\partial x_i \partial x_j)$ $(i = 1, \cdots, nx; \quad j = 1, \cdots, nx)$ of a function of many variables, $f(\boldsymbol{x})(\boldsymbol{x} = \{x_j\}; j = 1, \cdots, nx)$.

(2) **Usage**

Double precision:

ierr = ASL_dqmohx (f, x, nx, eps, mr, h, hes, iwk, wk);

Single precision:

ierr = ASL_rqmohx (f, x, nx, eps, mr, h, hes, iwk, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name of function f(x) that defines the function $f(\boldsymbol{x})$ of $\boldsymbol{x}$. (See Notes (a))
2	x	{D∗ / R∗}	nx	Input	Differentiation point $(x_1, \cdots, x_{nx})$.
3	nx	I	1	Input	Number nx of independent variables x
4	eps	{D / R}	1	Input	Required relative precision (Default value: (Unit for determining error)$^{\frac{4}{9}}$
5	mr	I	1	Input	Maximum number of iterations (Default value: 100)
6	h	{D / R}	1	Input	Initial step size
7	hes	{D∗ / R∗}	nx× nx	Output	Hessian $\partial^2 f/(\partial x_i \partial x_j)$ of the function at the differentiation point
8	iwk	I*	nx	Work	Work area
9	wk	{D∗ / R∗}	See Contents	Work	Work area (See Notes (b)) **Size**: $(\mathrm{nx} + \mathrm{mr} + 1) \times (5 + \mathrm{mr})$
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) eps $\geq \varepsilon \times 64$
(except when 0.0 is entered in order to set eps to the default value,
ε is the unit for determining error.)

(b) mr > 0
(except when 0.0 is entered in order to set mr to the default value)

(c) nx > 0

(d) h $\geq \max(x_i \times \varepsilon, \varepsilon^{0.25})$ $(i = 1, \cdots, \mathrm{nx})$
(ε is the unit for determining error.)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) or (b) was not satisfied.	Processing is performed with the corresponding default value set.
2000	The error became large before the required precision was achieved or the step size was too small.	The calculation value at that time is output, and processing is aborted.
2500	The maximum number of iterations was reached without the required precision being achieved.	
3000	Restriction (c) or (d) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) The function f should be created as follows.

```
double FORTRAN f(double *x)
{
    return(f[0],..., f[nx-1]);
}
```

(b) When reserving the area for the eighth argument wk as an array, let the size of the array be (nx + 101) $\times$ 105 if mr ≤ 0.

(c) Enter a value larger than the default value as the required relative precision.

(d) The value of the sixth argument h should be on the order of 0.5 to 1.0.

(e) For the convergence decision, consider that the sequence has converged when the following relationship holds:
{| Extrapolation value | $-$ | Previous extrapolation value |$\leq$| eps $\times$ Extrapolation value |}

(f) If a default value is shown in the Contents column of the table describing the arguments, the default value will be set if 0 is entered for an integer type argument or 0.0 is entered for a real type argument.

(g) Each elements of Hessian at differentiation point are stored in hes as shown below. hes[(i $-$ 1) + nx $*$ (j $-$ 1)] $= \partial^2 \mathrm{f}/\partial \mathrm{x_i} \partial \mathrm{x_j}$ $(i = 1, 2, \cdots, \mathrm{nx};\ j = 1, 2, \cdots, \mathrm{nx})$

(h) If several different errors occur at the same time, the most severe error value will be output for the error indicator, and the other error information may disappear.

(7) **Example**

(a) Problem

Obtain Hessian of the function $f(x_1, x_2, x_3) = x_1 x_2 x_3$ at $(x_1, x_2, x_3) = (5.5, 1.0, 8.0)$.

(b) Input data

Function name: f,

x[0]=5.5, x[1]=1.0, x[2]=8.0, nx=3, eps=1.0e−10, mr=15 and h=0.5.

(c) Main program

```
/*      C interface example for ASL_dqmohx */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return (x[0]*x[1]*x[2]);
}
#ifdef __cplusplus
}
#endif


int main()
{
    double *x1;
    int nx;
    double epsn;
    int mr;
    double h;
    double *ans;
    int *iwk;
    double *wk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dqmohx.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dqmohx ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=3;
    mr=15;
    x1  = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x1  == NULL )
    {
        printf( "no enough memory for array x1\n" );
        return -1;
    }
    ans  = ( double * )malloc((size_t)( sizeof(double) * (nx*nx) ));
    if( ans  == NULL )
    {
        printf( "no enough memory for array ans\n" );
        return -1;
    }
    wk  = ( double * )malloc((size_t)( sizeof(double) * ((nx+mr+1)*(5+mr)) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }
    iwk  = ( int * )malloc((size_t)( sizeof(int) * nx ));
    if( iwk  == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x1[i] );
```

```
    }
    fscanf( fp, "%lf", &epsn );
    fscanf( fp, "%lf", &h );
    printf( "\tNumber of Variable = %6d\n",nx );
    printf( "\n\tx   = " );
    for( i=0 ; i<nx ; i++ )
    {
        printf( "%8.3g ",x1[i] );
    }
    printf( "\n" );
    printf( "\n\teps = %8.3g\n",epsn );
    printf( "\tmr  =   %6d\n",mr );
    printf( "\th   = %8.3g\n",h );

    fclose( fp );

    ierr = ASL_dqmohx(f, x1, nx, epsn, mr, h, ans, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tHesse[i][j] =\n\n" );
    for( j=0 ; j<nx ; j++ )
    {
        for( i=0 ; i<nx ; i++ )
        {
        printf( "\t%8.3g", ans[j+nx*i] );
        }
        printf( "\n" );
    }

    free( x1 );
    free( ans );
    free( wk );
    free( iwk );

    return 0;
}
```

(d) Output results

```
*** ASL_dqmohx ***

 ** Input **

Number of Variable =      3

x   =      5.5        1        8

eps =    1e-10
mr  =       15
h   =      0.5

 ** Output **

ierr =      0

Hesse[i][j] =

       0           8           1
       8           0         5.5
       1         5.5           0
```

3.2.4 ASL_dqmojx, ASL_rqmojx
Jacobian of Multiple Function of Many Variables

(1) **Function**

ASL_dqmojx or ASL_rqmojx uses Richardson's extrapolation to obtain the Jacobian $J = \{\partial f_i/\partial x_j\}$ of multiple functions of many variables, $\boldsymbol{f}(\boldsymbol{x}) = \{f_i(x_j)\}(i = 1, \cdots, nf;\ j = 1, \cdots, nx)$.

(2) **Usage**

Double precision:

ierr = ASL_dqmojx (sub, x, nx, nf, eps, mr, h, rjac, iwk, wk);

Single precision:

ierr = ASL_rqmojx (sub, x, nx, nf, eps, mr, h, rjac, iwk, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	sub	–	–	Input	Name of function sub(x, nx, f, nf) that defines function $\boldsymbol{f}(\boldsymbol{x})$. (See Notes (a))
2	x	{D∗ / R∗}	nx	Input	Differentiation point $(x_1, \cdots, x_{nx})$.
3	nx	I	1	Input	Number nx of independent variables x
4	nf	I	1	Input	Number nf of simultaneous functions
5	eps	{D / R}	1	Input	Required relative precision (Default value: (Unit for determining error)$^{\frac{2}{3}}$
6	mr	I	1	Input	Maximum number of iterations (Default value: 100)
7	h	{D / R}	1	Input	Initial step size
8	rjac	{D∗ / R∗}	nf× nx	Output	The values of Jacobian $J = \{\partial f_i/\partial x_j\}$
9	iwk	I*	nf	Work	Work area
10	wk	{D∗ / R∗}	See Contents	Work	Work area (See Notes (b)) **Size**: nf × (5 + mr)
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) eps $\geq \varepsilon \times 64$
(except when 0.0 is entered in order to set eps to the default value, ε is the unit for determining error.)

(b) mr > 0
(except when 0.0 is entered in order to set mr to the default value)

(c) nx > 0 and nf > 0

(d) h $\geq \max(x_i \times \varepsilon, \varepsilon^{0.25})$ $(i = 1, \cdots, \mathrm{nx})$
(ε is the unit for determining error.)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) or (b) was not satisfied.	Processing is performed with the corresponding default value set.
2000	The error became large before the required precision was achieved or the step size was too small.	The calculation value at that time is output, and processing is aborted.
2500	The maximum number of iterations was reached without the required precision being achieved.	
3000	Restriction (c) or (d) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) The function f should be created as follows.

```
void FORTRAN sub(double *x, int *ns, double *f, int *nf)
{
    f[0] = f1(x1, ··· , xnx);
    f[1] = f2(x1, ··· , xnx);
    ⋮
    f[(*nf) − 1] = f*nf(x1, ··· , xnx);
}
```

(b) Each elements of Jacobian at differentiation point are stored in rjac as shown below. rjac$[(\mathrm{i}-1)+\mathrm{nf}*(\mathrm{j}-1)] = \partial \mathrm{f_i}/\partial \mathrm{x_j}$ $(i = 1, 2, \cdots, \mathrm{nf};\ j = 1, 2, \cdots, \mathrm{nx})$

(c) When reserving the area for the eighth argument wk as an array, let the size of the array be nf $\times$ 105 if mr ≤ 0.

(d) Enter a value larger than the default value as the required relative precision.

(e) The value of the sixth argument h should be on the order of 0.5 to 1.0.

(f) For the convergence decision, consider that the sequence has converged when the following relationship holds:
{| Extrapolation value | − | Previous extrapolation value |≤| eps × Extrapolation value |}

(g) If a default value is shown in the Contents column of the table describing the arguments, the default value will be set if 0 is entered for an integer type argument or 0.0 is entered for a real type argument.

(h) If several different errors occur at the same time, the most severe error value will be output for the error indicator, and the other error information may disappear.

(7) **Example**

(a) Problem

Obtain Jacobian of the following vector of many variables functions:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ f_3(\boldsymbol{x}) \\ f_4(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} x_1x_2x_3 \\ x_1 + x_2 + x_3 \\ x_1x_2 + x_2x_3 + x_3x_1 \\ x_1x_2x_3 + x_1x_2 + x_1 \end{bmatrix}$$

at $(x_1, x_2, x_3) = (6.8, 9.1, 3.4)$.

(b) Input data

Function name: f.

x[0]=6.8, x[1]=9.1, x[2]=3.4, nx=3, nf=4, eps=1.0e−8, mr=15 and h=0.5.

(c) Main program

```
/*      C interface example for ASL_dqmojx */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void   f(double *x,int *nx,double *f,int *nf)
#else
void   f(x,nx,f,nf)
double *x;
double *f;
int    *nx;
int    *nf;
#endif
{
    f[0]=x[0]*x[1]*x[2];
    f[1]=x[0]+x[1]+x[2];
    f[2]=x[0]*x[1]+x[1]*x[2]+x[2]*x[0];
    f[3]=x[0]*x[1]*x[2]+x[0]*x[1]+x[0];
}
#ifdef __cplusplus
}
#endif

int main()
{
    double *x1;
    int nx;
    int nf;
    double epsn;
    int mr;
    double h;
    double *ans;
    int *iwk;
    double *wk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dqmojx.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dqmojx ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=3;
    nf=4;
```

```
    mr=15;
    x1  = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x1  == NULL )
    {
        printf( "no enough memory for array x1\n" );
        return -1;
    }
    ans  = ( double * )malloc((size_t)( sizeof(double) * (nf*nx) ));
    if( ans  == NULL )
    {
        printf( "no enough memory for array ans\n" );
        return -1;
    }
    wk  = ( double * )malloc((size_t)( sizeof(double) * (nf*(5+mr)) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }
    iwk  = ( int * )malloc((size_t)( sizeof(int) * nf ));
    if( iwk  == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x1[i] );
    }
    fscanf( fp, "%lf", &epsn );
    fscanf( fp, "%lf", &h );
    printf( "\tNumber of Variable = %6d\n",nx );
    printf( "\tNumber of Function = %6d\n",nf );
    printf( "\n\tx   =" );
    for( i=0 ; i<nx ; i++ )
    {
        printf( " %8.3g ",x1[i] );
    }
    printf( "\n" );
    printf( "\n\teps = %8.3g\n",epsn );
    printf( "\tmr  =   %6d\n",mr );
    printf( "\th   = %8.3g\n",h );

    fclose( fp );

    ierr = ASL_dqmojx(f, x1, nx, nf, epsn, mr, h, ans, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tJacobi[i][j] =\n\n" );
    for( j=0 ; j<nf ; j++ )
    {
        for( i=0 ; i<nx ; i++ )
        {
        printf( "\t%8.3g", ans[j+nf*i] );
        }
        printf( "\n" );
    }

    free( x1 );
    free( ans );
    free( wk );
    free( iwk );

    return 0;
}
```

(d) Output results

```
*** ASL_dqmojx ***

 ** Input **

Number of Variable =      3
Number of Function =      4

x   =      6.8       9.1       3.4

eps =    1e-08
mr  =       15
h   =      0.3

 ** Output **

ierr =      0

Jacobi[i][j] =

    30.9        23.1        61.9
       1           1           1
    12.5        10.2        15.9
      41        29.9        61.9
```

Chapter 4

NUMERICAL INTEGRATION

4.1 INTRODUCTION

This chapter describes functions that integrate functions using automatic integration methods.
Although numerical integration can be performed using numerical table input or function input, this chapter deals only with functions using function input. For numerical table input, you can use functions described in Chapter 3, "Spline Functions."
When integrating a function, you can obtain high-precision solutions quickly by selecting the optimum solution method according to the function characteristics. This library provides functions corresponding the function characteristics as follows.

(1) Arbitrary functions
Functions for arbitrary functions quickly obtain the value of the integral for functions such as oscillatory functions, peak-type functions, functions having end-point singularities, and functions having interior-point singularities when you have no information about the characteristics of the function. These functions also output information related to the singular points. If the function has a considerable number of singularities or if it oscillates wildly making it impossible to obtain a high-precision solution, the functions calculate an approximate value and output it together with its error. If you want to obtain a high-precision solution, you should use other functions that use information about singular points. Although functions for arbitrary functions are the most general-purpose functions, they are unsuitable for oscillatory functions over an infinite interval.

(2) $f(x) \times w(x)$ type functions
Functions for $f(x) \times w(x)$ type functions obtain the value of the integral when either a trigonometric function, algebraic or logarithmic function having singularities at its endpoints or the function $\frac{1}{x-c}$ is used as the weight function $w(x)$.

(3) Oscillatory or peak-type functions
Use functions for oscillatory or peak-type functions if the function does not have a considerable number of singularities. You can increase computational efficiency by selecting the value of isw according to whether the function is an oscillatory or peak-type function.

(4) Singular function
You can use functions intended for singular functions to obtain the value of the integral even if the function has no singularities. These functions are more efficient than other functions, however, if the function has a considerable number of singularities. You should not use these functions for an oscillatory function. Functions are provided for when singular points are located at the end points and at interior points. You must provide information, however, about singular point positions when they are located at interior points.

(5) Singular functions for which singularity information is unknown
These functions, like the ones for arbitrary functions described in (1), obtain the value of the integral for any type of function when you have no information about function characteristics. These functions are suitable for functions having a more heavily singular point than the functions described in (1). In addition,

if the function is unable to reach the required precision, it may abort processing before completion without switching to processing to obtain an approximate solution like the functions described in (1) do. In addition, these functions require a great deal of calculation time. Therefore, you should use these functions if you want to obtain a very precise integral value when you know the function has singularities but you either do not know the positions of the singular points or you cannot determine them precisely.

4.1.1 Notes

(1) Figures 4−1 and 4−2 are flow diagrams that explain how to use the various functions introduced above.

Figure 4−1 Using Functions for a Finite Interval

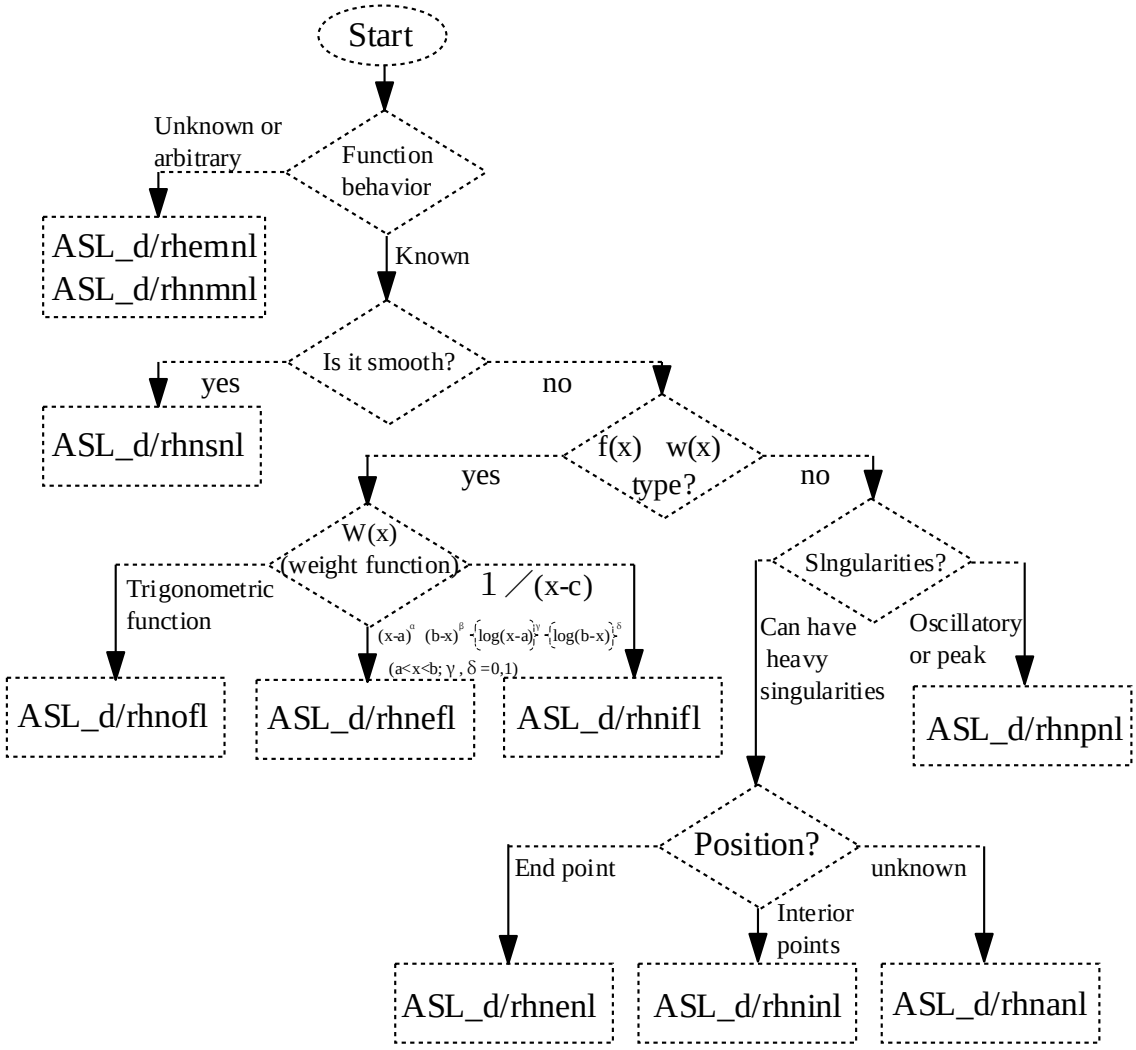
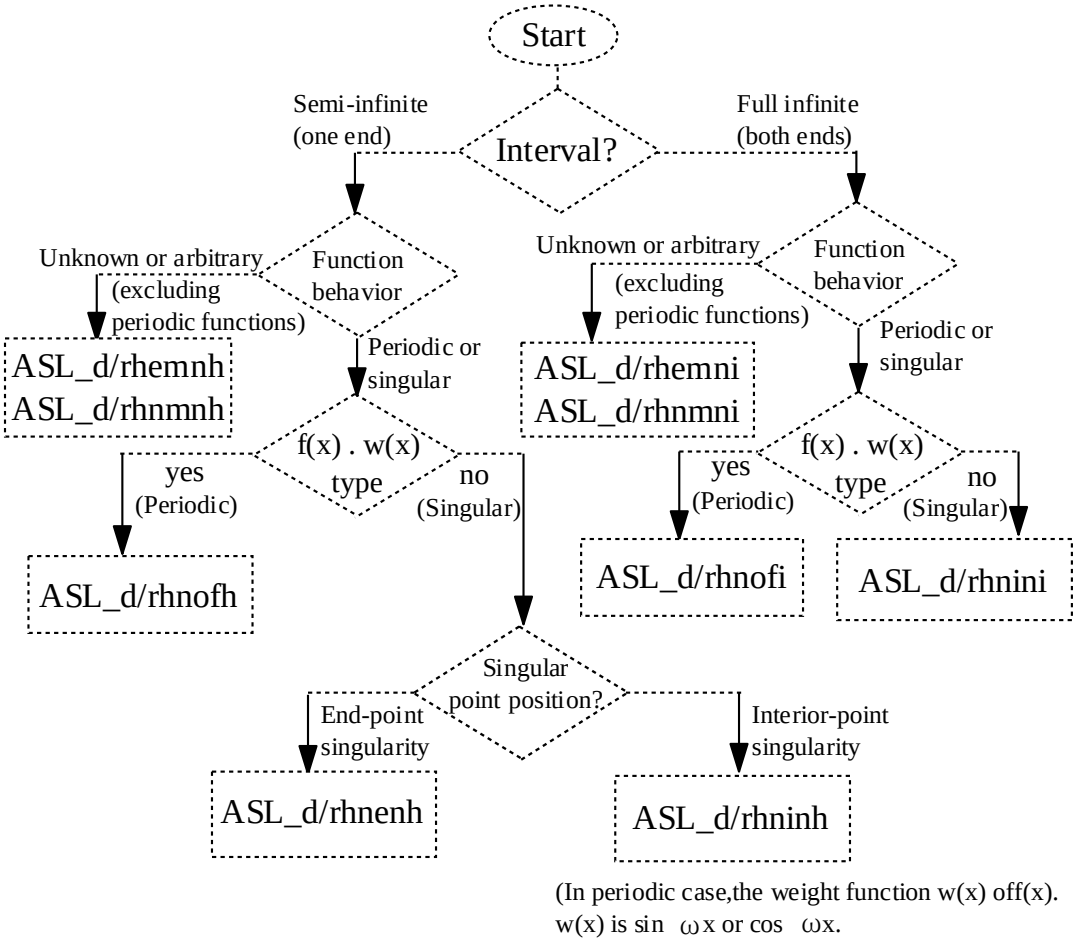


Figure 4−2 Using Functions for an Infinite Interval



(2) Note the following points concerning programs using these functions.

① You must take precautionary measures to prevent an overflow from occurring within the integration interval (such as setting the function value to 0.0 at singular points).

Example

One-dimensional integration (Let f has the same name in the all functions)

```
/* C interface example for ASL_dhemnl */
#include <stdio.h>
#include <math.h>
#include <asl.h>
```

- Function f

```
double FORTRAN f (double*x)
{
        }
```

if(fabs(∗x − 1.0) < uf)
    return 0.0;
else
    return 1.0/sqrt(fabs(∗x − 1.0));

} → { Measure to prevent overflow due to division by zero (unnecessary if overflow will not occur).

}

- Main function

```
int main()
{
    ≀
  ierr = ASL_dhemnl(f, ···);
    ≀
  return 0;
}
```

Example

Two-dimensional integration (Let f, a and b have the same names in the all functions)

```
/* C interface example for ASL_dhnfnm */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <asl.h>
```

- Function f

```
double FORTRAN f (double *x, double *y)
{
    ≀
}
```

- Function a

```
double FORTRAN a (double *y)
{
    ≀
}
```

- Function b

```
double FORTRAN b (double *y)
{
    ≀
}
```

- Main function

```
int main()
{
    ≀
  ierr = ASL_dhnfnm(f, a, b, ···);
    ≀
  return 0;
}
```

Example

Multi-dimensional integration (Let f and r have the same names in the all functions)

```
/* C interface example for ASL_dhnfml */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <asl.h>
```

- Function f

```
double FORTRAN f (double *x, int *m)
{
    ≀
  return Expression which variable is x[0], x[1], ···, x[*m − 1];
}
```

- Function r

```
void FORTRAN r (int *i, double *x, double *a, double *b, int *m)
{
    ≀
  switch(*i)
  {
    case 1 :
      a[0] = ···;
      b[0] = ···;   } ··· Expression which variable is x[1], x[2], ···, x[*m − 1].
    case 2 :
      a[1] = ···;
      b[1] = ···;   } ··· Expression which variable is x[2], ···, x[*m − 1].
    case 3 :
      ≀
    case *m :
      a[m − 1] = ···;
      b[m − 1] = ···;   } ··· Constant
```

```
    }
        ≀
}
```

- Main function

```
int main()
{
        ≀
    ierr = ASL_dhnfml(f, r, ···);
        ≀
    return 0;
}
```

(3) If several errors occur, the error number of the most severe error is output as the error indicator; other error information may be lost.

(4) These functions in this chapter calculate the value of the integral of functions to reach the required relative and/or absolute precision, which is given as an input argument. These functions obtain as output arguments the estimate value of absolute error as well as the integral value.

Notes Required relative precision, required absolute precision, and estimate value of absolute error are defined as follows:

Let us denote an numerical approximate solution of the integral value by Q and the precise solution by Q_0. Define an absolute error and a relative error by the following formulas:

$$\begin{cases} \text{(absolute error)} &= & |\, Q - Q_0 \,| \\ \text{(relative error)} &= & |\, (Q - Q_0)/Q \,| \end{cases}$$

Then, a required relative (absolute) precision ER (EA) means a tolerable upper limit for the relative (absolute) error, respectively. An estimate value of absolute error means a upper bound for an absolute error that is estimated by some numerical error estimation criterion.

When a required relative precision ER and/or required absolute precision EA is given as an input, these functions in this chapter will seek for an approximate solution of the integral value Q in such a precision as to satisfy the following condition(s):

$$\begin{cases} \text{AE}/Q &< & \text{ER} \\ \text{AE} &< & \text{EA} \end{cases}$$

(5) With all the functions in this chapter excluding the functions for multi-dimensional integration over a finite interval, the value used as the default value for required absolute precision is (minimum absolute value) $\times 2^{24}$.

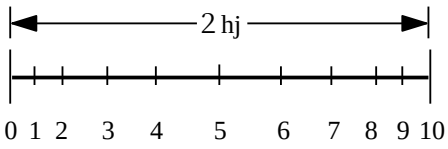
4.1.2 Algorithms Used

4.1.2.1 Adaptive Newton-Cotes rule (Integration of arbitrary functions)

The functions use an adaptive automatic integral method based on the Newton-Cotes 9-point rule and having additional functions for strengthening error estimation, easing convergence decisions, and detecting and processing singular points.
If $f(x)$ is assigned as the integrand $[a, b]$ as the integral interval, and ε_0 as the required absolute precision, then the automatic integral method automatically partitions the integral interval and calculates the integral value to a precision within the permitted error. The integral value and its error as well as a convergence decision are required on each subdivision interval in order to control this partitioning.
These values can be obtained as described below.

(1) Integral value and its error
For example, if the interval width, which is assumed to be $2h_j$, is partitioned by the 9-point rule as shown in Figure 4−3, then the integral value Q_j and its revised value ξ_j are expressed by equations (4.2) and (4.4).

Figure 4−3 Taking Interior Division Points (9-point Rule Example)



$$Q_j = \frac{h_j}{14175}\{989(f_0+f_{10})+5888(f_2+f_8)-928(f_3+f_7) \quad (4.1)$$
$$+10496(f_4+f_6)-4540f_5\} \quad (4.2)$$

$$\xi_j = \frac{-4736h_j}{468242775}\{3003(f_0+f_{10})-16384(f_1+f_9)+27720(f_2+f_8) \quad (4.3)$$
$$-38220(f_3+f_7)+56056(f_4+f_6)-64350f_5\} \quad (4.4)$$

If we use the Newton-Cotes n-point rule and assume that the $(n+1)$-th differential coefficient is almost fixed, then the error ε_j for $Q_j+\xi_j$ is $\frac{|\,\xi_j\,|}{(2^{n+1}-1)}$.
However, the $(n+1)$-th and subsequent differential coefficients cannot actually be ignored and often become larger than this.
The following method is used to further subdivide a given interval i into two parts obtaining intervals j and $j+1$ and to estimate ε_j from ξ_i and ξ_j. If we assume that the ratio of $\varepsilon+|\,\xi\,|$ and $|\,\xi\,|$ is fixed, then:

$$C = \frac{|\xi_j+\xi_{j+1}|}{|\xi_i|}$$

(If the $(n+1)$-th differential coefficient is fixed, then $C = \frac{1}{2^{n+1}}$.) If the function characteristics are almost identical in the neighboring interval, we can assume that $\xi_j = \xi_{j+1}$. From this we obtain:

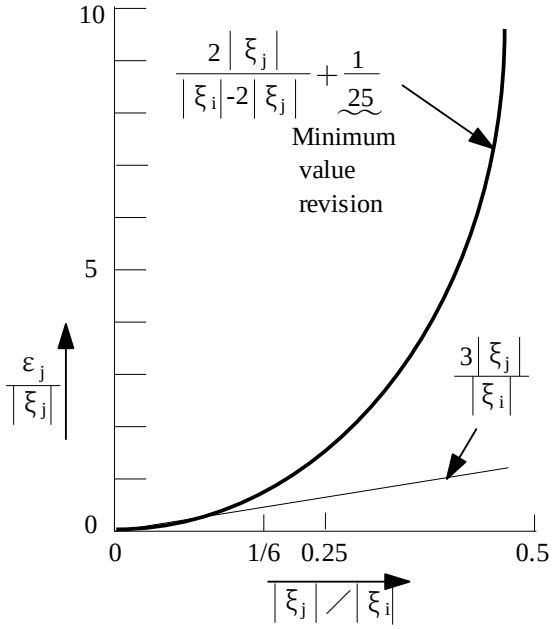
$$\varepsilon_j = \frac{C}{1-C}\,|\,\xi_j\,| = \frac{2{\xi_j}^2}{|\,\xi_i\,|-2\,|\,\xi_j\,|} \quad (4.5)$$

The sum of these ε_j values for all intervals becomes the entire amount of error.
In the program, to keep the error estimate from being too small where the minimum value of ε_j is bounded

by $\frac{|\xi_j|}{25}$ (for single precision, $\frac{|\xi_j|}{8}$) and to prevent the error from being excessively evaluated in an area having $|\xi_i| \leq 6 |\xi_j|$ and a small reduction rate ξ such as in the neighborhood of a singular point, the following relationships are assumed:
$\frac{1}{1-C} = 1.5$ and $\varepsilon_j = 3C\frac{|\xi_j|}{2} = 3\frac{{\xi_j}^2}{|\xi_i|}$. (See Figure 4−4) In addition, for stability, a large value is estimated for ε_j involved in the convergence decision by taking $\varepsilon_j = \frac{|\xi_j|}{2.0}$.

If the value actually taken for $\frac{|\xi_j|}{|\xi_i|}$ exceeds 0.25, then the series generally does not converge and the point

Figure 4−4 Revising ε_j Values



should be processed as a singular point. In its neighborhood, however, the value of $\frac{|\xi_j|}{|\xi_i|}$ may fluctuate dramatically or the $\frac{|\xi_j|}{|\xi_i|}$ convergence-time value may be close to 1.0. In this case, since the estimate of the error ε_j is too large, the error is approximated by the dashed line in Figure 4−4 for $\frac{|\xi_j|}{|\xi_i|} \geq \frac{1}{6}$ to revise the error for $\frac{|\xi_j|}{|\xi_i|} = 0.25$.

(2) Method of determining convergence in each interval
Let $h_0 = \frac{b-a}{2}$, and $h_j = \frac{(\text{Width of subdivision interval } j)}{2}$ for $(j = 1, \cdots, n)$. If we assume that the error for a solution that is considered to have converged has a uniform probability density $g(x)$ on the interval $[-\varepsilon_j, \varepsilon_j]$, then $g(x) = \frac{1}{2\varepsilon_j}$ and the variance in this interval at this time is $\frac{{\varepsilon_j}^2}{3}$. Therefore, the variance for the entire interval is $\sum_{j=1}^{n} \frac{{\varepsilon_j}^2}{3}$. If we assume $\varepsilon_j = \sqrt{\frac{h_j}{h_0}}\varepsilon_0$ (where ε_0 is the required absolute precision of the integral on $[a, b]$), then the variance for the integral value over the entire interval is $\sigma^2 = \frac{{\varepsilon_0}^2}{3}$. Therefore, the error becomes the normal distribution $\sigma = \frac{\varepsilon_0}{\sqrt{3}}$, and the probability that the value is within the range from $-\varepsilon_0$ through ε_0 becomes 91.6 percent. Therefore, if we let ε_j be the value $\frac{|\xi_j|}{2.0}$ obtained in (a), partitioning

ends when the following inequality holds and $Q_j + \xi_j$ is the integral value over the given interval.

$$\frac{|\,\xi_j\,|}{2.0} < \sqrt{\frac{h_j}{h_0}}(\max(\varepsilon_0, \varepsilon_0' I))$$

ε_0': Required relative precision of the integral value over $[a, b]$

I: Estimate of total integral value

[If Q_1 is the integral value obtained by the 9-point Newton-Cotes rule, the $I = Q_1 + \xi_1 + \xi_2 + \cdots + \xi_n + q$ (Revision for singular point processing.)]

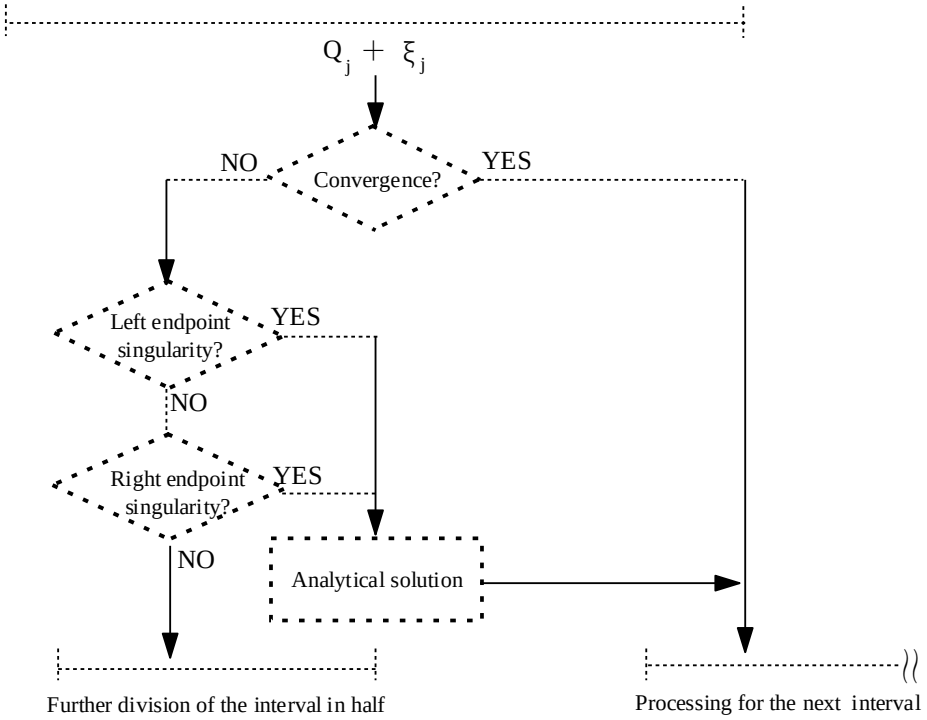
(3) Processing if a singular point is at a subdivision point

If a singular point is located at one of the points that divides the total interval into 2^m equal parts for a suitable positive integer m, then that singular point will become an endpoint of the interval at some step within the evaluation process.

Function values at interior points are used to detect singular points according to the flow shown in Figure 4−5. The detection operation occurs at this time when the interval width becomes $2^{-\ell}$ (where ℓ is a multiple of 5) times the width of the total interval.

The following equations are used for singular point detection and processing.

Figure 4−5 Calculation Procedure



- Algebraic singularity ($f(x) = \alpha X^{-p} + q$)

  Left endpoint singularity

  The following equation holds:

  $$\frac{f_2 - f_3}{f_3 - f_5} \approx \frac{f_3 - f_5}{f_5 - f_{10}} = d$$

  Analytical solution I_j

  $$I_j = \frac{2h_j(f_{10} - pq)}{1 - p}, \quad p = \frac{\log(d)}{\log(2)}, \quad q = \frac{2^{-p} f_5 - f_{10}}{2^{-p} - 1}$$

Right endpoint singularity
The following equation holds:

$$\frac{f_8 - f_7}{f_7 - f_5} \approx \frac{f_7 - f_5}{f_5 - f_0} = d$$

Analytical solution I_j

$$I_j = \frac{2h_j(f_0 - pq)}{1 - p}, \quad p = \frac{\log(d)}{\log(2)}, \quad q = \frac{2^{-p}f_5 - f_0}{2^{-p} - 1}$$

- Logarithmic singularity
  Left endpoint singularity
  The following equation holds:

$$f_2 - f_3 \approx f_3 - f_5 \approx f_5 - f_{10} = d$$

  Analytical solution I_j

$$I_j = 2h_j \left(f_5 + d\frac{1 - \log(2)}{\log(2)} \right)$$

  Right endpoint singularity
  The following equation holds:

$$f_8 - f_7 \approx f_7 - f_5 \approx f_5 - f_0 = d$$

  Analytical solution I_j
  Same equation as for the left endpoint singularity.

For both algebraic and logarithmic singularities, the almost equal expressions $\approx$ are considered to hold when the left and right sides of the expression differ by δ, where $\delta \leq \frac{\varepsilon_0'}{\sqrt{10h_j/h_0}}$ (ε_0' : Required relative precision). In this case, singular point processing is performed. In addition, the error e_j is assumed to be:
$e_j =$| (| δ | +Units for determining error) $\times I_j$ |

(4) Processing for singular points not detected by the method described in (3)
If the singular point was not detected by the method described in (3), then a value approximated by the algebraic singularity $y = \alpha x^p$ $(0 < p < 1)$ is used. That is, k is assumed to be the point where the maximum absolute value is taken. Then $f_k = 0$ is assumed, and:

$$\delta_s = E_s \frac{p + 1}{(2 - 2^{-p})(1 - p)}$$

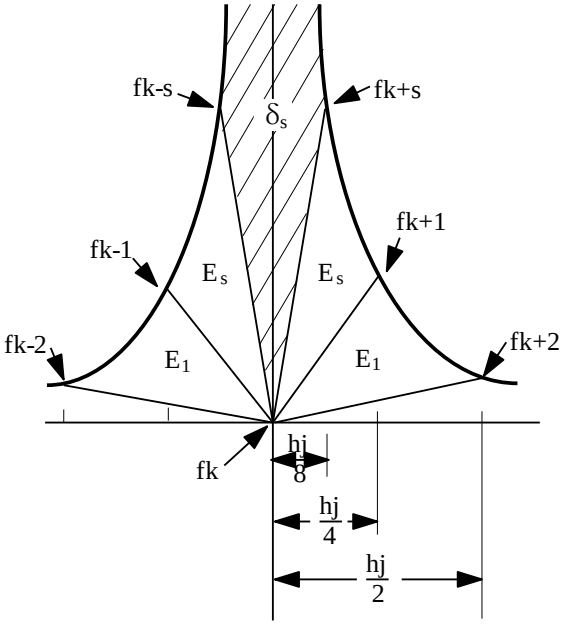
(This holds for either an endpoint or interior point.)

$$p = \frac{\log(2E_s/E_1)}{\log(2)}$$

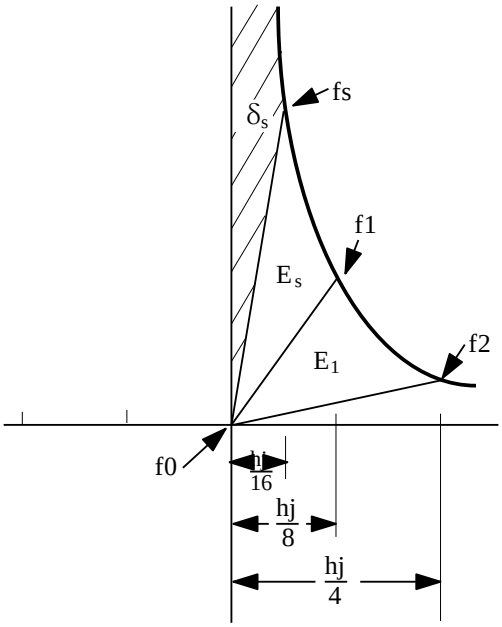
$$\left[\begin{array}{l} E_1 \text{ and } E_s \text{ are analytically calculated from function values.} \\ f_{k+s} \text{ and } f_{k-s} \text{ are function value used to calculate } \delta_s \\ \text{(See Figure 4–6(a), (b))} \end{array} \right]$$

Figure 4−6 Singular Point Processing

(a) Interior singular point



(b) End singular point



The integral value is calculated by adding δ_s obtained analytically to the integral value obtained by the trapezoidal rule for the portion where δ_s could not be added. In addition, the error is assumed to be the difference between the δ_s portion and the value obtained by the trapezoidal rule.

For singular points that cannot be detected by the methods described in (3) and (4), you can obtain a value that is as close as possible to the true value by an algorithm that improves the Aitken extrapolation method.

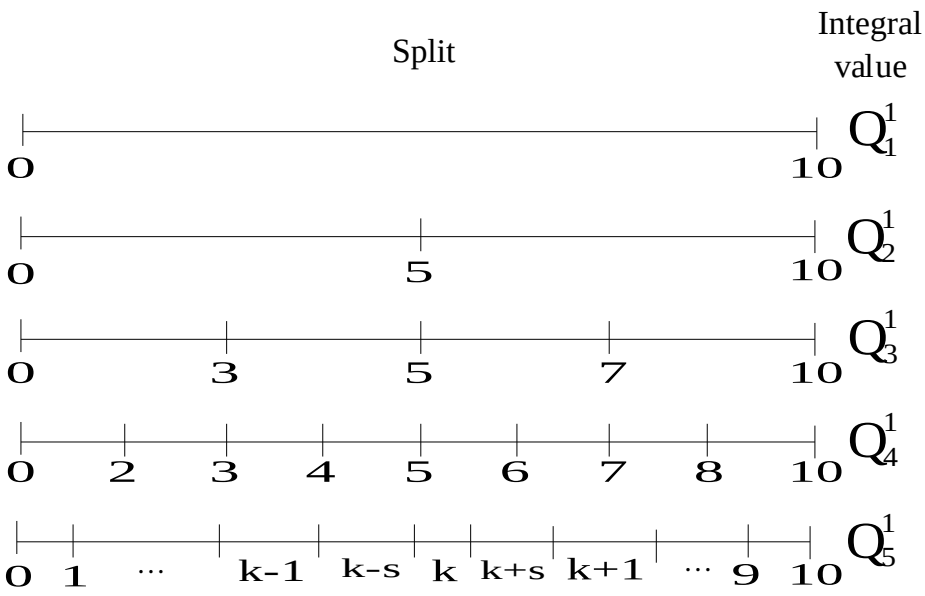
- Aitken extrapolation improvement

  Let the integral value for each subdivision be Q_n^1 for $(n = 1, \cdots, 5)$ as shown in Figure 4−7.

  Use the extrapolation formula shown below to create a table of Q_n^k for $(k = 1, 2, 3; n = 2k-1, \cdots, 5)$.

$$Q_n^k = Q_n^{k-1} - \frac{(Q_n^{k-1} - Q_{n-1}^{k-1})^2}{Q_n^{k-1} - 2Q_{n-1}^{k-1} + Q_{n-2}^{k-1}}$$

Figure 4−7 Various Subdivisions and Their Integral Values



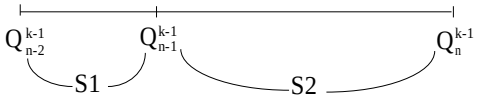
However, if $| Q_n^{k-1} - Q_{n-1}^{k-1} | \geq | Q_{n-1}^{k-1} - Q_{n-2}^{k-1} |$ then Q_n^k is Obtained from $Q_n^k = \dfrac{Q_n^{k-1} + mQ_{n-1}^{k-1}}{m+1}$ (m : Weight).

Finally, set integral value to Q_5^3 and the error is set to be $| Q_5^3 - Q_5^1 |$

Remarks

a. The weight is obtained from the following calculation.

$$m = \frac{| Q_n^{k-1} - Q_{n-1}^{k-1} |}{| Q_{n-1}^{k-1} - Q_{n-2}^{k-1} |} \frac{1}{15}$$



(If $S2$ is large compared to $S1$, then the weight on the Q_n^{k-1} side is decreased and a value close to Q_{n-1}^{k-1} is returned).

(5) Applications for an infinite interval

Use a variable transformation. Calculate the integrals over the interval $[0, 1]$ using the transformation shown below.

$$\begin{aligned}
\int_a^\infty f(x)dx &= \int_0^1 \frac{F(t)}{t^2}dt \\
F(t) &= f(a + \frac{1-t}{t}) \\
\int_{-\infty}^\infty f(x)dx &= \int_0^1 \frac{F(t) + G(t)}{t^2}dt \\
F(t) &= f(\frac{1-t}{t}), \quad G(t) = f(\frac{t-1}{t})
\end{aligned}$$

(6) Applications for a double integral
To evaluate the double integral:

$$I = \int_c^d \int_a^b f(x, y)dxdy$$

create a function that obtains $F(y)$ defined as:

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

and calculate:

$$\int_c^d F(y)dy$$

while calling this function.
If the integral area is not rectangular, to evaluate the double integral:

$$I = \int_c^d \int_{a(y)}^{b(y)} f(x, y)dxdy$$

create a function that obtains $F(y)$ defined as:

$$A = a(y), \quad B = b(y), \quad F(y) = \int_A^B f(x, y)dx$$

and calculate:

$$\int_c^d F(y)dy$$

while calling this function.

4.1.2.2 Gauss-Kronrod Method

A problem when using the group of Gauss formulas is that the previously calculated function values may not be reused when the dimension has been increased since the sampling points are the zeroes of a spherical polynomial. Therefore, Kronrod added $(n+1)$ points to the n-point Gauss rule to create a $(3n+1)$ degree integral rule based on $(2n+1)$ points. However, since the n-point Gauss rule weight is not maintained, a new weight must be used for the $(2n+1)$ points.
The non-adaptive functions start from the 10-point Gauss rule and modify integral values according to a Kronrod rule that increases the number of points to 21, 43, 87 or 175 points. If the differences between these modified values do not exceed the required precision, then the series is considered to have converged, and the updated value is returned as the integral value.
In a manner similar to the adaptive Newton-Cotes rule, the adaptive automatic integral functions continue to subdivide the integral interval until the error is within the allowable precision while adding the integral values over each subdivided interval to obtain the total integral value. However, in this case, the integral value and error are calculated by using formulas for Gauss-Kronrod pairs corresponding to 7-15 points, 10-21 points, 15-31 points, 20-41 points, 25-51 points or 30-61 points. In addition if there is no singularity information, the function extrapolates the integral value of the ε-algorithm based on the 10-21 point formula.
If a trigonometric function or an algebraic or logarithmic function is used for the weight function, then adaptive

automatic integration is performed for the 7-15 point pair for the portions away from the singularities or where there is little oscillation, but the Clenshaw-Curtis rule is applied to the intervals containing singularities or where the function oscillates wildly.
To estimate the error e_j in interval j using the Gauss-Kronrod method, use the following equations:

$$\begin{aligned} e'_j &= \int_j \mid f(x) - \frac{Q_j}{h_j} \mid dx \\ e_j &= e'_j \min\left\{1.0, \frac{\xi_j}{e'_j}\right\} \end{aligned}$$

The values Q_j and ξ_j are the integral value and its revised value, respectively, in the interval j.

4.1.2.3 Clenshaw-Curtis method (functions having a weight function)

The Clenshaw-Curtis method approximates the integrand $f(x)$ by a Chebyshev polynomial of the form $\sum_{i=0}^{N}(a_i T_i(x))$, integrates this expansion to obtain the Chebyshev expansion $\sum_{i=0}^{N}(b_i T_i(x))$ corresponding to the integral value $Q(x) = \int_{-1}^{1} f(x)dx$, and obtains the integral value from this integrated expansion.
Expanding $f(x)$ as described above, we obtain:

$$f(x) = \frac{1}{2}a_0 + a_1 T_1(x) + a_2 T_2(x) \ + \cdots$$

The coefficients a_k, which are the Fourier-Chebyshev coefficients, are given by the following formula.

$$\begin{aligned} a_k &= \frac{2}{\pi}\int_{-1}^{1}\frac{f(x)T_k(x)}{(1-x^2)^{\frac{1}{2}}}dx \\ &= \frac{2}{\pi}\int_{0}^{\pi} f(\cos(\theta))\cos(k\theta)d\theta \end{aligned}$$

Using the trapezoidal rule to approximate the calculation of a_k, we get:

$$\begin{aligned} a_k \approx \alpha_k &= \frac{1}{N}\left(f(x_0)T_k(x_0) + 2\sum_{j=1}^{N-1}(f(x_j)T_k(x_j)) + f(x_N)T_k(x_N)\right) \\ &= \frac{1}{N}\left(f(1) + 2\sum_{j=1}^{N-1}(f(\cos(\frac{\pi j}{N}))\cos(\frac{\pi k j}{N})) + (-1)^k f(-1)\right) \end{aligned}$$

If we truncate the polynomial for $f(x)$ at the N-th term, we get:

$$\begin{aligned} f(x) &\approx \frac{1}{2}a_0 + a_1 T_1(x) + a_2 T_2(x) + \cdots + a_N T_N(x) \\ &\approx \frac{1}{2}\alpha_0 + \alpha_1 T_1(x) + \cdots + \alpha_{N-1}T_{N-1}(x) + \frac{1}{2}\alpha_N T_N(x) \end{aligned}$$

The coefficients α_k can be obtained by a fast Fourier transform (FFT). In particular, Tolstov (1962) showed a simple processing method for $d = 12$. This library use $d = 12$ and $d = 24$.
When the weight function $w(x)$ is assigned to the function, the integration method is as follows.

$$\begin{aligned} &\int_{-1}^{1} w(x)f(x)dx \\ &= \frac{\alpha_0}{2}\int_{-1}^{1} w(x)dx + \sum_{k=1}^{N-1}\alpha_k \int_{-1}^{1}\{w(x)T_k(x)\}dx + \frac{\alpha_N}{2}\int_{-1}^{1}\{w(x)T_N(x)\}dx \end{aligned}$$

- When $w(x)$ is $\sin(\omega x)$

$$\begin{aligned}\int_{c_1}^{c_2} w(x)f(x)dx \quad \approx \quad & \frac{1}{2}(c_2-c_1)\Bigg[\frac{\alpha_0}{\lambda}\sin\Big(\frac{(c_1+c_2)\omega}{2}\Big)\sin(\lambda) \\ & + \sum_{k=1}^{N-1}\alpha_k\Bigg\{\cos\Big(\frac{(c_1+c_2)\omega}{2}\Big)\int_{-1}^{1}\sin(\lambda x)T_k(x)dx \\ & + \sin\Big(\frac{(c_1+c_2)\omega}{2}\Big)\int_{-1}^{1}\cos(\lambda x)T_k(x)dx\Bigg\} \\ & + \frac{\alpha_N}{2}\Bigg\{\cos\Big(\frac{(c_1+c_2)\omega}{2}\Big)\int_{-1}^{1}\sin(\lambda x)T_N(x)dx \\ & + \sin\Big(\frac{(c_1+c_2)\omega}{2}\Big)\int_{-1}^{1}\cos(\lambda x)T_N(x)dx\Bigg\}\Bigg] \\ & \lambda = \frac{(c_2-c_1)\omega}{2}\end{aligned}$$

If we let $\int_{-1}^{1}\sin(\lambda x)T_k(x)dx$ be $S_k(\lambda)$ and $\int_{-1}^{1}\cos(\lambda x)T_k(x)dx$ be $C_k(\lambda)$, then $S_k(\lambda)$ and $C_k(\lambda)$ are expressed by the following recursive relations.

$$\begin{aligned}&\lambda^2(k-1)(k-2)S_{k+2}(\lambda) \; - \; 2(k^2-4)(\lambda^2-2k^2+2)S_k(\lambda) \; + \; \lambda^2(k+1)(k+2)S_{k-2}(\lambda) \\ &\quad = -8(k^2-4)\sin(\lambda) - 24\lambda\cos(\lambda)\end{aligned}$$

$$\begin{aligned}S_1(\lambda) &= 2(\sin(\lambda)-\lambda\cos(\lambda))\lambda^{-2} \\ S_3(\lambda) &= \lambda^{-2}\sin(\lambda)(18-48\lambda^{-2}) + \lambda^{-1}\cos(\lambda)(48\lambda^{-2}-2)\end{aligned}$$

$$\begin{aligned}&\lambda^2(k-1)(k-2)C_{k+2}(\lambda) \; - 2(k^2-4)(\lambda^2-2k^2+2)C_k(\lambda) \; + \; \lambda^2(k+1)(k+2)C_{k-2}(\lambda) \\ &\quad = 24\lambda\sin(\lambda) - 8(k^2-4)\cos(\lambda)\end{aligned}$$

$$\begin{aligned}C_0(\lambda) &= 2\lambda^{-1}\sin(\lambda) \\ C_2(\lambda) &= 8\lambda^{-2}\cos(\lambda) - \lambda^{-3}(2\lambda^2-8)\sin(\lambda) \\ C_4(\lambda) &= 32\lambda^{-4}(\lambda^2-12)\cos(\lambda) + 2\lambda^{-5}(\lambda^4-80\lambda^2+192)\sin(\lambda)\end{aligned}$$

$$S_{2k}(\lambda) \quad = \quad C_{2k+1}(\lambda) = 0 \;\; \text{for} \;\; (k=0,1,2\cdots)$$

These values can be similarly calculated when $w(x)$ is $\cos(\omega x)$.
If $\lambda \leq 2$, it is more efficient to use the Gauss-Kronrod rule than this method.

- When $w(x)$ is the function which has algebraic or logarithmic end-point singularities
If we let weight function be $u(x)$,

$$\begin{aligned}u(x) = (x-a)^{\gamma}(b-x)^{\delta}\{\log(x-a)\}^{\mu}\{\log(b-x)\}^{\nu} \\ (a \leq c_1 < c_2 \leq b, \mu, \nu = 0 \text{ or } 1)\end{aligned}$$

When c_1 is end-point and $c_1 = a$,

$$\int_a^{c_2} u(x)f(x)dx \approx \Big(\frac{c_2-a}{2}\Big)^{\gamma+1}\sum_{i=0}^{\mu}\{\log(c_2-a)\}^{\mu-i} \times\Big[\frac{\alpha_0}{2}\int_{-1}^{1}(1+x)^{\gamma}\Big\{\log\Big(\frac{1+x}{2}\Big)\Big\}^{i}dx+\sum_{k=1}^{N-1}\alpha_k G_{k,i}(\gamma)+\frac{\alpha_N}{2}G_{N,i}(\gamma)\Big]$$

here

$$G_{k,i}(\gamma)=\int_{-1}^{1}(1+x)^{\gamma}\left\{\log(\frac{1+x}{2})\right\}^{i}T_k(x)dx \quad \text{for} \quad (k=1,\cdots,N; i=0,1).$$

When c_2 is end-point and $c_2 = b$

$$\int_{c_1}^{b} u(x)f(x)dx \approx \Big(\frac{b-c_1}{2}\Big)^{\delta+1}\sum_{i=0}^{\nu}\{\log(b-c_1)\}^{\nu-i} \times\Big[\frac{\alpha_0}{2}\int_{-1}^{1}(1+x)^{\delta}\Big\{\log\Big(\frac{1+x}{2}\Big)\Big\}^{i}dx+\sum_{k=1}^{N-1}\alpha_k H_{k,i}(\delta)+\frac{\alpha_N}{2}H_{N,i}(\delta)\Big]$$

here

$$H_{k,i}(\delta)=(-1)^k G_{k,i}(\delta) \quad \text{for} \quad (k=1,\cdots,N; i=0,1).$$

- When $w(x)$ has interior point singularities for $\dfrac{1}{x-c}$

  If we let $v(x)=\dfrac{1}{x-c}$, then:

$$\int_{c_1}^{c_2} v(x)f(x)dx \approx \frac{\alpha_0}{2}\log|\frac{c'-1}{c'+1}|+\sum_{k=1}^{N-1}(\alpha_k V_k(c'))+\frac{\alpha_N}{2}V_N(c')$$

where:

$$c_1 < c < c_2$$
$$V_k(c')=\oint_{-1}^{+1}\frac{T_k(x)}{x-c'}dx, \qquad c'=\frac{2c-c_2-c_1}{c_2-c_1}$$

The values $V_k(c')$ are obtained by solving the following recursive relations.

$$V_{k+1}(c')-2c'V_k(c')+V_{k-1}(c')=\begin{cases}0 & k:\text{Odd Number}\\ 4 & \\ \dfrac{}{1-k^2} & k:\text{Even Number}\end{cases}$$

$$V_0(c') = \log|\frac{1-c'}{1+c'}|$$
$$V_1(c') = 2+c'V_0(c')$$

An adaptive-type integration is performed so that no subdivision is done at point c.

4.1.2.4 ε-algorithm

For a given series a_n for $(n = 0, 1, \cdots)$ let:

$$\begin{aligned} {\tau_n}^{(-1)} &= 0 \text{ for } (n = 0, 1, 2 \cdots) \\ {\tau_n}^{(0)} &= a_n \text{ for } (n = 0, 1, 2 \cdots) \end{aligned}$$

and starting form these values, sequentially generate the values ${\tau_n}^{(1)}, {\tau_n}^{(2)}, \cdots$ for $(k = 1, 2, 3, \cdots)$ from the equation:

$${\tau_n}^{(k)} = {\tau_{n-1}}^{(k-2)} + \frac{1}{{\tau_n}^{(k-1)} - {\tau_{n-1}}^{(k-1)}} \text{ for } (n = k, k+1, \cdots)$$

If we create the series, ${\tau_n}^{(2)}, {\tau_n}^{(6)}, {\tau_n}^{(8)}, \cdots$ in this way, it will converge to the value a_m $(m = \infty)$ faster than a_n.

$$\begin{array}{cccc} {\tau_1}^{(0)} & & & \\ {\tau_2}^{(0)} & {\tau_2}^{(1)} & & \\ {\tau_3}^{(0)} & {\tau_3}^{(1)} & {\tau_3}^{(2)} & \\ {\tau_4}^{(0)} & {\tau_4}^{(1)} & {\tau_4}^{(2)} & {\tau_4}^{(3)} \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

If you have no singularity information or if you are integrating an oscillatory function this method is used to obtain an approximate solution that is closer to the true solution by extrapolating the approximate integral solution for each step using the adaptive Gauss-Kronrod 10-21 point method or 7-15 point method. In addition, when integrating an oscillatory function over an infinite interval, it is used to extrapolate from the integral values over the number of finite intervals extending over the function cycle. (If you apply the ε-algorithm when you have no singularity information or when integrating an oscillatory function, updating proceeds by further dividing the smallest subdivided interval in half.)

4.1.2.5 Double exponential formula (integrating a function having endpoint or interior-point singularities)

The double exponential formula is an effective method of integral a function having singularities at the endpoints of the integration interval or for integrating over a semi-infinite or fully infinite interval. It obtains the integral value by converting to the function $f(\phi(u))\phi'(u)$ over the fully infinite interval $(-\infty, \infty)$, which is to be quickly reduced by a variable transformation, and applying the trapezoidal rule on a suitably truncated range. Values are further updated by dividing the step size in half until the integral value reaches the required precision. This method is described below.

(1) Let the initial step size be $h_0 = 0.25$ and let

$$W = f(\phi(0))\phi'(0)$$

(2) For $n = 1, 2, 3, \cdots$, let

$$W_n = W_{n-1} + f(\phi(nh_0))\phi'(nh_0) + f(\phi(-nh_0))\phi'(-nh_0)$$

and truncate additional terms when $| W_n - W_{n-1} | < \max(| W_n | \times$(Required relative precision), (Required absolute precision) $/(\frac{\pi}{2}h_0)) \times 0.01$
Let the truncated endpoints be $-Nh_0$ and Mh_0 and let $Q_0 = h_0 W_n$.

(3) Let the step size h_i be $\frac{1}{2}h_{i-1}$, and obtain integral values Q_i for $(i = 1, \cdots, 10)$ by the trapezoidal rule for the interval $[-Nh_0, Mh_0]$. In addition, let $Q_i - Q_{i-1} = \xi_i$.
When the following relationship holds:

$$\frac{4{\xi_i}^2}{\mid \xi_{i-1} \mid - \mid \xi_i \mid} + \mid \xi_i \mid < \max(\zeta_r Q_i, \zeta_a)$$

(ζ_r : Required relative precision, ζ_a : Required absolute precision)

Then, Q_i is returned as the integral value and $\frac{4{\xi_i}^2}{\mid \xi_{i-1} \mid - \mid \xi_i \mid} + \xi_i$ is returned as its absolute error.

(4) Estimate the integral value outside both the endpoints truncated in (b) by algebraically extrapolating from the values of terms added last and last but one, add the estimated value to the integral value, and then add the absolute value to the absolute error value.
The convergence determination method and error calculation formula carefully consider integration of a singular function. That is formula (4) of the adaptive Newton-Cotes rule is adapted to the non-adaptive trapezoidal rule to obtain $\varepsilon_i = \frac{{\xi_i}^2}{\mid \xi_{i-1} \mid - \mid \xi_i \mid}$.
If the convergence rate C always is fixed, the value of ε_i can be used to determine convergence. However, in reality, the convergence rate is not fixed and a series is assumed to have converged when the value of ε_i actually is a smaller value. Therefore, to estimate the error larger than the true error and to determine convergence safely, the calculation result for ε_i is multiplied by 4.0 to determine a safe rate, and the error is estimated by further adding $\mid \xi_i \mid$. This enables you to avoid obtaining a mistaken convergence (the actual error is larger than the estimated error and the calculation terminates before the required precision has been reached) according to the general method in which the error is assumed to be $\mid \xi_i \mid$.
If no canceling prevention transformation is performed in integration over a finite interval, then the function value is inaccurate if nh is large (about 2.5 for single precision or about 3.3 for double precision), $\phi(nh) \approx 1.0$, and there is a singularity within the range of ± 1.0 from the endpoints of $f(\phi(nh))$. Therefore, even if the integration after conversion has been obtained accurately, the conversion-time error is large, and this must be carefully considered in the integral value error. This program estimates error from the fact that convergence of (c) gets slower as the conversion-time error increases and adds this estimated error value to the estimated value for absolute error.
The convergence determination or error estimation methods described above can be used to integrate a function having no singularities without any problem other than the error estimate is a little large.
The value $\phi(nh)$ used here is as follows.
When $\int_{-1}^{1} f(x)dx$

$$\begin{aligned} x_n &= \phi(nh) = \tanh(\frac{\pi}{2}\sinh(nh)) \\ Q_h &= \frac{\pi}{2}h \sum_{n=-\infty}^{\infty} f(\tanh(\frac{\pi}{2}\sinh(nh)))\frac{\cosh(nh)}{\cosh^2(\pi/2\sinh(nh))} \end{aligned}$$

When $\int_{0}^{\infty} f(x)dx$

$$\begin{aligned} x_n &= \phi(nh) = \exp(\frac{\pi}{2}\sinh(nh)) \\ Q_h &= \frac{\pi}{2}h \sum_{n=-\infty}^{\infty} f(\exp(\frac{\pi}{2}\sinh(nh)))\cosh(nh)\exp(\frac{\pi}{2}\sinh(nh)) \end{aligned}$$

When $f(x)$ in $\int_0^\infty f(x)dx$ contains a factor of the form $\exp(-x)$

$$\begin{aligned} x_n &= \phi(nh) = \exp(\frac{\pi}{2}(nh - \exp(-nh))) \\ Q_h &= \frac{\pi}{2}h \sum_{n=-\infty}^{\infty} f(\exp\frac{\pi}{2}(nh - \exp(-nh)))(1 + \exp(-nh)) \exp\frac{\pi}{2} \\ & \quad (nh - \exp(-nh)) \end{aligned}$$

To obtain the integral value for a function having an interior-point singularity, the interval is divided at the singular point, integral values are obtained over each interval using the method described above for a function having an endpoint singularity, and the integral values are added together to obtain the total integral value.

Remarks

a. For canceling prevention

Canceling occurs if the denominator of $f(x)$ in the integral $\int_{-1}^{1} f(x)dx$ includes a factor of $(1+x)^\alpha, (1-x)^\beta$ $(0 < \alpha, \beta < 1)$. Therefore, we let $u = nh$ and the integral is divided so that:

- For $-1 \le x < 0$

$$\begin{aligned} 1 + x &= 1 + \tanh(\frac{\pi}{2}\sinh(u)) \\ &= \exp(\frac{\pi}{2}\sinh(u))/\cosh(\frac{\pi}{2}\sinh(u)) = -t_1 \end{aligned}$$

- For $0 \le x \le 1$

$$\begin{aligned} 1 - x &= 1 - \tanh(\frac{\pi}{2}\sinh(u)) \\ &= \exp(-\frac{\pi}{2}\sinh(u))/\cosh(\frac{\pi}{2}\sinh(u)) = t_2 \end{aligned}$$

and

$$\begin{aligned} \int_{-1}^{1} f(x)dx &= \int_{-1}^{0} f(x)dx + \int_{0}^{1} f(x)dx \\ &= \int_{-1}^{0} f(-1-t_1)dt_1 + \int_{0}^{1} f(1-t_2)dt_2 \end{aligned}$$

4.1.2.6 Integrating an oscillatory function over an infinite interval

If we assume the weight function is $\cos(\omega x)$ or $\sin(\omega x)$, the integrand that includes the weight function is $f(x)$, the period for which $f(x) = 0$ is λ, the phase shift is θ, and an arbitrary integer is m, then:

$$Q = \int_0^\infty f(x)dx, \quad f(m\lambda + \theta) = 0$$

Now, we assume M is a sufficiently large positive number and consider the following transformation:

$$x = M\phi(t), \quad \phi(-\infty) = 0, \quad \phi(+\infty) = 0$$

If we perform this transformation, the original integral is expressed as follows:

$$Q = \int_{-\infty}^{\infty} f(M\phi(t))M\phi'(t)dt$$

If we evaluate this by applying the trapezoidal rule with step-size h, we obtain the following expression:

$$Q \approx Mh \sum_{n=-\infty}^{\infty} f(M\phi(nh + \frac{\theta}{M}))\phi'(nh + \frac{\theta}{M}) \tag{4.6}$$

where we assume $\phi(t)$ is a function that satisfies the following:

$$\lim_{t\to\infty} \phi(t) = t$$

Then, if we take h so that $Mh = \lambda$, the following will hold for a large value of n :

$$f(M\phi(nh + \frac{\theta}{M})) \approx f(Mnh + \theta) = f(n\lambda + \theta) = 0$$

If we choose the following functions as $\phi(t)$ and $\phi'(t)$:

$$\begin{cases} \phi(t) &= \dfrac{t}{1 - \exp(-K \sinh t)} \\ \\ \phi'(t) &= \dfrac{1 - (1 + Kt \cosh t) \exp(-K \sinh t)}{(1 - \exp(-K \sinh t))^2} \end{cases} \qquad (\text{Assume } \ K = 6)$$

then as $t \to +\infty$, $\phi(t)$ approaches t and $f(M\phi(t)) \approx 0$, and as $t \to -\infty$, $\phi'(t)$ approaches 0. Also, since digits will be lost if this calculation is performed with t close to zero, the following approach is used.
If $\mid t \mid$ is less than the square root of the unit for determining error, then we choose the following functions as $\phi(t)$ and $\phi'(t)$.

$$\begin{cases} \phi(t) &= \dfrac{1}{K} \\ \phi'(t) &= 0.5 \end{cases}$$

The integration procedure is as follows.
Let the initial step size h in equation (4.6) be

$$h = \frac{3.23}{-\log((\text{Required absolute precision}))}$$

The summational interval range in equation (4.6) is expanded by 1 for both side until to satisfy below equation from Richardson's extrapolate equation and integral value is obtained.

$$\max\left\{\frac{t_1}{15}, \frac{19t_1 - t_2}{45}\right\} \times Mh < \frac{(\text{Required absolute precision})}{16}$$

where t_1 is sum of $f(M\phi(nh + \dfrac{\theta}{M}))\phi'(nh + \dfrac{\theta}{M})$ at current interval range both end-point and t_2 is a value of t_1 at previous step.
Let obtained integral value be S_1, halve the step size h repeatedly and obtain the integral value $S_2, S_3 \cdots$ similarly. When satisfy

$$\mid S_n - S_{n-1} \mid \leq \sqrt{\frac{(\text{Required absolute precision})}{10}}$$

let S_n be the integral value.

4.1.2.7 Multi-dimensional integration over a finite interval

First a function is numerically integrated in a dimension using the formula of the Gauss-Romberg rule with N sample points, and the obtained result is denoted by G_N. Next the formula is applied to numerically integrate it in the next dimension. This process is repeated for the successive dimensions. Using the Cartesian product and the product formula, the whole process is expressed as $G_N \times G_N \times \cdots$.
If the value of N is increased as $N_{n+1} = N_n + 2, N_1 = 2 \ \text{ for } \ (n = 1, 2, \cdots)$, a series which approaches the true solution can be obtained. This series is accelerated with the following θ-algorithm, and an approximate solution is obtained.

θ-algorithm
Set $\theta_{-1}^{(n)} = 0, \theta_0^{(n)} = S^{(n)}$ ($S^{(n)}$ is the series for an approximate solution). Then

$$\theta_{2k+1}^{(n)} = \theta_{2k-1}^{(n+1)} + \frac{1}{\Delta\theta_{2k}^{n}}$$

Approximate solution

if not ($|\ \Delta\theta_{2k}^{(n)}\ | > |\ \Delta\theta_{2k}^{(n+1)}\ | > |\ \Delta\theta_{2k}^{(n+2)}\ |$) then

  if $\left(|\ \Delta\theta_{2k}^{(n+1)}\ | > |\ \Delta\theta_{2k}^{(n+2)}\ |\right)$ then

    $\theta_{2k+2}^{(n)} = \theta_{2k}^{(n+2)} + \dfrac{1}{\Delta\theta_{2k+1}^{(n+1)}}$ *

  else if $\left(\mathrm{sign}(\Delta\theta_{2k}^{(n+2)}) = \mathrm{sign}(\Delta\theta_{2k}^{(n+1)})\right)$ then ($\mathrm{sign}(x)$ gives a sign of value x)

    $\theta_{2k+2}^{(n)} = \theta_{2k}^{(n+2)} + 1.5\Delta\theta_{2k}^{(n+1)}$ **

  else

    $\theta_{2k+2}^{(n)} = \dfrac{\theta_{2k}^{(n+3)} + \theta_{2k}^{(n+1)}}{2}$ **

  end

else if $\left(|\ \Delta\theta_{2k+1}^{(n+1)}\ | \leq |\ \Delta\theta_{2k+1}^{(n)}\ |\right)$ then

  $\theta_{2k+2}^{(n)} = \theta_{2k}^{(n+2)} + \dfrac{1}{\Delta\theta_{2k+1}^{(n+1)}}$ *

else

  $\theta_{2k+2}^{(n)} = \theta_{2k}^{(n+1)} + \dfrac{\Delta\theta_{2k}^{(n+1)}\Delta\theta_{2k+1}^{(n+1)}}{\Delta^2\theta_{2k+1}^{(n)}}$

end

The estimated error in the approximate integration is

$$\varepsilon = \left\{\begin{matrix}6 \text{ in double precision} \\ 9 \text{ in single precision}\end{matrix}\right\} \times \max(|\ \theta_{2k+2}^{(n)} - \theta_{2k}^{(n+2)}\ |, |\ \theta_{2k+2}^{(n)} - \theta_{2k}^{(n+3)}\ |)$$

for the processing without $*$, twice ε for those with $*$, and 10 times ε for those with $**$.

4.1.2.8 Integral of the product of arbitrary function and special functions

(1) Definite integral of the product of Chebyshev polynomial and Bessel function of the order 0 $\displaystyle\int_0^1 J_0(\alpha x)T_n(2x-1)xdx$

From the Chebyshev expansion expression of $f(x)$

$$f(x) = \sum_{n=0}^{\infty} C_n T_n(2x-1)$$

the definite integral $\int_0^1 J_0(\alpha x)f(x)xdx$ is calculable as

$$\sum_{n=0}^{\infty} C_n \int_0^1 J_0(\alpha x)T_n(2x-1)xdx$$

by using applying the definite integral $\int_0^1 J_0(\alpha x)T_n(2x-1)xdx$.
Here, in $\alpha \leq 15.0$, it is more stable to use Gauss = Legendre's integration formula directly.

1. Obtaining $W_{0,1} = \int_0^1 J_0(\alpha x)dx$.
Using

$$\int_0^1 J_0(\alpha x)dx = (2N+1)^{-1}\sum_{k=0}^{2N}\frac{\sin(\alpha\cos(2\pi k/(2N+1)))}{\alpha\cos(2\pi k/(2N+1))}.$$

2. Obtaining $I_n = \int_0^1 J_0(\alpha x)T_n(2x-1)xdx$
It will be set to

$$I_n = J_1(\alpha)/\alpha + 2n^2 J_0(\alpha)/\alpha^2 - \alpha^{-2}(W_{n,3} + W_{n,2})$$

if $J_0(\alpha x)$ is expressed with this differentiation and the second degree differentiation using the differential equation of the 0th Bessel function and changing by integration by parts where

$$W_{n,1} = \int_0^1 J_0(\alpha x)T_n(2x-1)dx$$

$$W_{n,2} = \int_0^1 J_0(\alpha x)(T_n(2x-1))''xdx$$

$$W_{n,3} = \int_0^1 J_0(\alpha x)(T_n(2x-1))'dx$$

Furthermore, the relations

$$W_{0,1} = \int_0^1 J_0(\alpha x)dx$$

$$W_{0,2} = W_{0,3} = W_{1,2} = 0$$

$$W_{1,3} = 2W_{0,1}, W_{1,1} = 2I_0 - W_{0,1}, W_{2,2} = 16I_0, W_{2,3} = 16I_0 - 8W_{0,1}$$

$$W_{n,1} + 2W_{n-1,1} + W_{n-2,1} = 4I_{n-1}(n \geq 2)$$

$$W_{n,3}/n - W_{n-2,3}/(n-2) = 4W_{n-1,1}(n \geq 3)$$

$$W_{n,2}/n - W_{n-2,2}/(n-2) = (n-1)/nW_{n,3} + 2W_{n-1,3} + (n-1)/(n-2)W_{n-2,3}(n \geq 3)$$

which are obtained from the properties of Chebyshev polynomial $T_n(x)$, are used.

(2) Definite integral of the product of arbitrary function $f(x)$ and the 0th Bessel function $\int_0^1 J_0(\alpha x)f(x)xdx$
Dividing the interval [0,1] to small intervals, approximating $f(x)$ as the polynomial $P(x^2)$ in each interval $[\alpha_1, \alpha_3]$ with the condition that the values of $P(x)$ at the points $\alpha_j^2(j = 1, 2, 3)$ are $f(\alpha_j)$ and the degree of $P(x)$ is two or less, where α_2 is the middle point of each interval, calculating $\int_{\alpha_1}^{\alpha_3} J_0(\alpha x)P(x^2)xdx$ analytically, adds over all the subdivision intervals.

(3) Infinite integration $\int_{-\infty}^{\infty} e^{-x^2}f(x)dx$
$\int_{-\infty}^{\infty} e^{-x^2}f(x)dx$ can be approximated as

$$\sum_{1\leq j\leq n, -3\leq m\leq 3} w_j e^{-(z_j+2m)^2}f(z_j+2m)$$

$(z_j, w_j(j = 1, 2, \cdots, n)$ are Gauss points (zero points of Legendre polynomial of the degree n) and weights).

4.1.3 Reference Bibliography

(1) Fritsch, F. N. , Kahaner, D. K. and Lyness, J. N. , "Double Integration Using One-Dimensional Adaptive Quadrature Routines: A Software Interface Problem", ACM Trans Math. Softw. Vol. 7, pp.46-75, (1979).

(2) Patterson, "The Optimal Addition of Points to Quadrature Formulae", Math. Comp. Vol.22, (1968).

(3) Piessens and Branders, "A Note of the Optimal Addition of Abscissas to Quadrature Formulas of Gauss and Lobatto Type", Math. Comp. Vol.28, (1974).

(4) Wynn, "On the Convergence and Stability of the Epsilon Algorithm", J. SIAM Num. Anal. Vol.3, No.1, (1966).

(5) Monegato, "A Note on Extended Gaussian Quadrature Rules", Math. Comp. Vol.30, (1976).

4.2 INTEGRATION OVER A FINITE INTERVAL

4.2.1 ASL_dhemnl, ASL_rhemnl
Arbitrary Function

(1) **Function**

ASL_dhemnl or ASL_rhemnl automatically integrates the function over a finite interval. Even if the integrand has singularities, ASL_dhemnl or ASL_rhemnl determines its characteristics and automatically processes it. These functions are easy to use since the number of required input arguments has been minimized.

(2) **Usage**

Double precision:

```
ierr = ASL_dhemnl  (f, a, b, er, &q, &ae);
```

Single precision:

```
ierr = ASL_rhemnl  (f, a, b, er, &q, &ae);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	er	{D / R}	1	Input	Required relative precision (Default value : Unit for determining error ×64) (See Notes (c)) (See Section 4.1.1)
5	q	{D∗ / R∗}	1	Output	Integral value
6	ae	{D∗ / R∗}	1	Output	Estimate value of absolute error (See Section 4.1.1)
7	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	There are five or more singular points.	Processing continues.
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by -1 or the integral value $= 0.0$.
1500	Restriction (b) was not satisfied.	Processing is performed with er set to the default value.
2000	The interval was subdivided 500 times.	The minimum subdivision width is gradually widened so that an approximate solution can be obtained.
2400	A certain subdivided interval cannot be further subdivided.	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3500	The result is unreliable (the error is larger than the result.)	Processing is aborted.
4000	Overflow occurred more than once in a single subdivided interval.	

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integration interval. (For example, you can set the function value at singular points to 0.0.)

(b) If the integrand has an interior-point singularity, you should obtain the solution by having the singular point become one of the points dividing the entire interval into 2^n equal parts. In addition, if the required precision for double-precision calculations is not more lenient than the default value, the solution precision may worsen and output for the number of singular points may be greater than the actual number.
If the integrand has numerous peaks, you should increase the required precision by using the double-precision function when obtaining the solution.
At other times or if you have no information about function characteristics, you should specify the precision you require as the required precision when obtaining the solution.

(c) If you input 0.0 for the variable er, the default value is set.

(d) This function uses an algorithm that is based on the adaptive Newton-Cotes 9-point rule but having more powerful singular point processing capabilities.

(7) **Example**

(a) Problem

Obtain the value of $\int_0^1 \sqrt{x} \log x dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

(Assume f=0.0 when $x = 0.0$.)

a=0.0, b=1.0 and er=0.0.

(c) Main program

```
/*      C interface example for ASL_dhemnl */

#include <stdio.h>
#include <math.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
        return sqrt(*x) * log(*x);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double epsrel;
    double result;
    double abserr;
    int ierr;
    FILE *fp;

    fp = fopen( "dhemnl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhemnl ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &epsrel );

    printf( "\ta   = %8.3g\n", a );
    printf( "\tb   = %8.3g\n", b );
    printf( "\ter  = %8.3g\n", epsrel );

    fclose( fp );

    ierr = ASL_dhemnl(f, a, b, epsrel, &result, &abserr);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );

    return 0;
}
```

(d) Output results

```
*** ASL_dhemnl ***

 ** Input **

a   =         0
b   =         1
er  =         0

 ** Output **

ierr =      0

Integral Approximation
  q   =   -0.444

Estimate of Absolute Error
  ae  =  1.6e-15
```

4.2.2 ASL_dhnsnl, ASL_rhnsnl Smooth Function

(1) **Function**

ASL_dhnsnl or ASL_rhnsnl integrates a smooth function having no singularities over a finite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnsnl  (f, a, b, er, ea, &q, &ae, &nev);
```

Single precision:

```
ierr = ASL_rhnsnl  (f, a, b, er, ea, &q, &ae, &nev);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
6	q	{D∗ / R∗}	1	Output	Integral value
7	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
8	nev	I*	1	Output	Number of times integrand is evaluated
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea $\geq$ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by -1 or the integral value $= 0.0$
1500	Restriction (b) or (c) was not satisfied.	Processing is performed with er or ea set to the default value.
2500	The solution precision will not reach the required precision	Processing is terminated without obtaining a solution having the required precision.
3500	The result is unreliable (the error is larger than the result).	Processing is aborted.

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set 0.0 is input.

(b) This function uses a non-adaptive algorithm that starts with the 10-point Gauss rule and then updates the integral value according to the Kronrod rule in which the number of points is increased to 21, 43, 87 or 175 points.

(7) **Example**

(a) Problem
Obtain the value of $\int_0^1 (x^2 - 2x + 1)dx$.

(b) Input data
Function name corresponding to integrand $f(x)$: f.
a=0.0, b=1.0, er=0.0 and ea=0.0.

(c) Main program

```
/*      C interface example for ASL_dhnsnl */

#include <stdio.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return ((*x)*(*x) - 2.0*(*x) + 1.0) ;
}
#ifdef __cplusplus
}
#endif


int main()
{
```

```
    double a;
    double b;
    double ers;
    double eas;
    double q;
    double ae;
    int nev;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnsnl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnsnl ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &ers );
    fscanf( fp, "%lf", &eas );

    printf( "\ta   = %8.3g\n", a );
    printf( "\tb   = %8.3g\n", b );
    printf( "\ter  = %8.3g\n", ers );
    printf( "\tea  = %8.3g\n", eas );

    fclose( fp );

    ierr = ASL_dhnsnl(f, a, b, ers, eas, &q, &ae, &nev);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", q );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", ae );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", nev );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnsnl ***

 ** Input **

a   =        0
b   =        1
er  =        0
ea  =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =    0.333

Estimate of Absolute Error
  ae  = 1.48e-16

Number of Function Evaluations
  nev =     21
```

4.2.3 ASL_dhnofl, ASL_rhnofl
Function of the Type $f(x)\cdot$ ($\sin\omega x$ or $\cos\omega x$)

(1) **Function**

ASL_dhnofl or ASL_rhnofl integrates an oscillatory function that can be factored into $f(x)\cdot$ ($\sin\omega x$ or $\cos\omega x$) over a finite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnofl  (f, a, b, w, itype, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnofl  (f, a, b, w, itype, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the factor of integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	w	{D / R}	1	Input	ω of the weight function ($\sin\omega x$, $\cos\omega x$)
5	itype	I	1	Input	Weight function type $1\cdots\int f(x)\cos(\omega x)dx$ $2\cdots\int f(x)\sin(\omega x)dx$
6	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error $\times 64$) (See Section 4.1.1)
7	ea	{D / R}	1	Input	Required relative precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
8	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value:500)
9	q	{D∗ / R∗}	1	Output	Integral value
10	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
11	nev	I*	1	Output	Number of times integrand is evaluated
12	iwk	I*	2×idv	Work	Work area (a size of 1000 should be reserved if zero is entered for idv)
13	wk	{D∗ / R∗}	4×idv	Work	Work area (a size of 2000 should be reserved if zero is entered for idv)
14	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) idv > 1 (except when 0 is input since the default value is assumed)

(e) itype =1 or 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by −1 or the integral value= 0.0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv.	Processing is terminated without obtaining a solution having the required precision.
2400	A certain subdivided interval cannot be further subdivided.	
2500	The solution precision will not reach the required precision.	
2700	Solutions obtained according to the $\varepsilon-$algorithm did not converge.	
3000	Restriction (e) was not satisfied.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0.0 is input.

(b) This function uses a 25-point modified Clenshaw-Curtis rule to integrate over intervals where the function oscillates wildly and combine this with the 13-point rule to calculate its error. In places where the function does not oscillate wildly, these functions calculate the integral value and its error according to the 7-15 point Gauss-Kronrod rule.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_0^{\frac{\pi}{2}} \sin x \cdot \frac{1}{\sqrt{1-0.25\sin^2 x}}dx$.

(b) Input data

Function name corresponding to factor of integrand $f(x)$: f.

a=0.0, b=$\frac{\pi}{2}$, w=1.0, itype=2, er=0.0, ea=0.0 and idv=0.

(c) Main program

```
/*      C interface example for ASL_dhnofl */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return 1.0/(sqrt(1.0-0.25*sin(*x)*sin(*x)));
}
#ifdef __cplusplus
}
#endif


int main()
{
    double a;
    double b;
    double w;
    int itype;
    double er;
    double ea;
    int idv;
    int idv0=500;
    double q;
    double ae;
    int nev;
    int *iwk;
    double *wk;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnofl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnofl ***\n" );
    printf( "\n     ** Input **\n\n" );
    idv=0;
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &w );
    fscanf( fp, "%d", &itype);
    fscanf( fp, "%lf", &er );
    fscanf( fp, "%lf", &ea );

    iwk = ( int * )malloc((size_t)( sizeof(int) * (2*idv0) ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (4*idv0) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\tw     = %8.3g\n", w );
    printf( "\titype =   %6d\n", itype );
    printf( "\ter    = %8.3g\n", er );
    printf( "\tea    = %8.3g\n", ea );
    printf( "\tidv   =   %6d\n", idv );

    fclose( fp );
```

```
    ierr = ASL_dhnofl(f, a, b, w, itype, er, ea, idv, &q, &ae, &nev, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", q );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", ae );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", nev );

    free( iwk );
    free(  wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnofl ***

 ** Input **

a     =        0
b     =     1.57
w     =        1
itype =        2
er    =        0
ea    =        0
idv   =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =      1.1

Estimate of Absolute Error
  ae  = 1.05e-14

Number of Function Evaluations
  nev =     75
```

4.2.4 ASL_dhnefl, ASL_rhnefl

Function of the Type $f(x) \cdot ((x-a)^{\alpha}(b-x)^{\beta}\{\log(x-a)\}^{\gamma}\{\log(b-x)\}^{\delta})(a < x < b; \gamma, \delta = 0, 1)$

(1) **Function**

ASL_dhnefl or ASL_rhnefl integrates a function having an endpoint singularity that can be factored into $f(x) \cdot ((x-a)^{\alpha}(b-x)^{\beta}\{\log(x-a)\}^{\gamma}\{\log(b-x)\}^{\delta})(a < x < b; \gamma, \delta = 0, 1)$ over a finite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnefl  (f, a, b, alfa, beta, itype, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnefl  (f, a, b, alfa, beta, itype, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the factor of integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	alfa	{D / R}	1	Input	α of the weight function $(x-a)^{\alpha}(\alpha>-1)$
5	beta	{D / R}	1	Input	β of the weight function $(b-x)^{\beta}(\beta>-1)$
6	itype	I	1	Input	Weight function logarithm factor type $=1:(x-a)^{\alpha}\cdot(b-x)^{\beta}$ $=2:(x-a)^{\alpha}\cdot(b-x)^{\beta}\cdot\log(x-a)$ $=3:(x-a)^{\alpha}\cdot(b-x)^{\beta}\cdot\log(b-x)$ $=4:(x-a)^{\alpha}\cdot(b-x)^{\beta}\cdot\log(x-a)\cdot\log(b-x)$
7	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
8	ea	{D / R}	1	Input	Required absolute precision (Default value: positive minimum value $\times 2^{24}$) (See Section 4.1.1)
9	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value:500)
10	q	{D∗ / R∗}	1	Output	Integral value
11	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
12	nev	I*	1	Output	Number of times integrand is evaluated
13	iwk	I*	idv	Work	Work area (a size of 1000 should be reserved if zero is entered for idv)
14	wk	{D∗ / R∗}	4 × idv	Work	Work area (a size of 2000 should be reserved if zero is entered for idv)
15	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) idv > 1 (except when 0 is input since the default value is assumed)

(e) itype = 1, 2, 3 or 4

(f) alfa > −1.0 and beta > −1.0

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by −1 or the integral value= 0.0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv.	Processing is terminated without obtaining a solution having the required precision.
2400	A certain subdivided interval cannot be further subdivided.	
2500	The solution precision will not reach the required precision.	
3000	Restriction (e) or (f) was not satisfied.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integration interval. (For example, you can set the function value at singular points to 0.0).

(b) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(c) This function uses a 13-point and 25-point modified Clenshaw-Curtis rule to calculate the integral value and its error over the subdivided interval that includes the end-point and a 7-15 point Gauss-Kronrod rule over other subdivided intervals.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_0^1 \frac{\log(\frac{1}{x})}{\sqrt{x}}dx$.

(b) Input data

Function name corresponding to factor of integrand $f(x)$: f.

(Assume f=0.0 when $x=0.0$.)

a=0.0, b=1.0, alfa=-0.5, beta=0.0, itype=1, er=0.0, ea=0.0 and idv=0.

(c) Main program

```
/*      C interface example for ASL_dhnefl */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
        return log(1.0/(*x));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double alfa;
    double beta;
    int itype;
    double er;
    double ea;
    int idv;
    int idv0=500;
    double q;
    double ae;
    int nev;
    int *iwk;
    double *wk;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnefl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnefl ***\n" );
    printf( "\n     ** Input **\n\n" );
    idv=0;
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &alfa );
    fscanf( fp, "%lf", &beta );
    fscanf( fp, "%d", &itype);
    fscanf( fp, "%lf", &er );
    fscanf( fp, "%lf", &ea );

    iwk = ( int * )malloc((size_t)( sizeof(int) * idv0 ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (4*idv0) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\talfa  = %8.3g\n", alfa );
```

```
    printf( "\tbeta  = %8.3g\n", beta );
    printf( "\titype =   %6d\n", itype );
    printf( "\ter    = %8.3g\n", er );
    printf( "\tea    = %8.3g\n", ea );
    printf( "\tidv   =   %6d\n", idv );

    fclose( fp );

    ierr = ASL_dhnefl(f, a, b, alfa, beta, itype, er, ea, idv, &q, &ae, &nev, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", q );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", ae );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", nev );

    free( iwk );
    free(  wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnefl ***

 ** Input **

a     =        0
b     =        1
alfa  =     -0.5
beta  =        0
itype =        1
er    =        0
ea    =        0
idv   =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =        4

Estimate of Absolute Error
  ae  =  1.1e-13

Number of Function Evaluations
  nev =   4490
```

4.2.5 ASL_dhnifl, ASL_rhnifl
Function of the Type $f(x) \cdot (1/(x-c))$

(1) **Function**

ASL_dhnifl or ASL_rhnifl integrates a function having an interior-point singularity that can be factored into $f(x) \cdot (1/(x-c))$ over a finite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnifl  (f, a, b, c, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnifl  (f, a, b, c, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the factor of integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	c	{D / R}	1	Input	c of the weight function $1/(x-c)$
5	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error $\times 64$) (See Section 4.1.1)
6	ea	{D / R}	1	Input	Required absolute precision (Default value: Unit for determining error $\times 2^{24}$) (See Section 4.1.1)
7	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value: 500)
8	q	{D∗ / R∗}	1	Output	Integral value
9	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
10	nev	I*	1	Output	Number of times integrand is evaluated
11	iwk	I*	idv	Work	Work area (a size of 1000 should be reserved if zero is entered for idv)
12	wk	{D∗ / R∗}	$4 \times$ idv	Work	Work area (a size of 2000 should be reserved if zero is entered for idv)
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) idv > 1 (except when 0 is input since the default value is assumed)

(e) c ≠ a or b

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by −1 or the integral value = 0.0
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv.	Processing is terminated without obtaining a solution having the required precision.
2400	A certain subdivided interval cannot be further subdivided.	
2500	The solution precision will not reach the required precision.	
3000	Restriction (e) was not satisfied.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(b) This function uses a 13-point and 25-point modified Clenshaw-Curtis rule to calculate the integral value and its error over the subdivided interval that includes the point c and a 7-15 point Gauss-Kronrod rule over other subdivided intervals.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_{-1}^{5} \frac{1}{x(5x^3+6)}$.

(b) Input data

Function name corresponding to factor of integrand $f(x)$: f.

a=−1.0, b=5.0, c=0.0, er=1.0e−8, ea=0.0 and idv=0.

(c) Main program

```
/*      C interface example for ASL_dhnifl */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return 1.0/(5.0*(*x)*(*x)*(*x)+6.0);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double c;
    double er;
    double ea;
    int idv;
    int idv0=500;
    double q;
    double ae;
    int nev;
    int *iwk;
    double *wk;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnifl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnifl ***\n" );
    printf( "\n     ** Input **\n\n" );
    idv=0;
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &c );
    fscanf( fp, "%lf", &er );
    fscanf( fp, "%lf", &ea );

    iwk = ( int * )malloc((size_t)( sizeof(int) * idv0 ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (4*idv0) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\tc     = %8.3g\n", c );
    printf( "\ter    = %8.3g\n", er );
```

```
    printf( "\tea    = %8.3g\n", ea );
    printf( "\tidv   =   %6d\n", idv );

    fclose( fp );

    ierr = ASL_dhnifl(f, a, b, c, er, ea, idv, &q, &ae, &nev, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", q );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", ae );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", nev );

    free( iwk );
    free(  wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnifl ***

 ** Input **

a     =       -1
b     =        5
c     =        0
er    =    1e-08
ea    =        0
idv   =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =  -0.0899

Estimate of Absolute Error
  ae  = 3.51e-11

Number of Function Evaluations
  nev =    355
```

4.2.6 ASL_dhnpnl, ASL_rhnpnl
General Oscillatory or Peak-Type Function

(1) **Function**

ASL_dhnpnl or ASL_rhnpnl integrates a function having a weak singularity over a finite interval. You must determine the value of isw corresponding to the singularity type.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnpnl  (f, a, b, er, ea, idv, &q, &ae, &nev, isw, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnpnl  (f, a, b, er, ea, idv, &q, &ae, &nev, isw, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
6	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value:500)
7	q	{D∗ / R∗}	1	Output	Integral value
8	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
9	nev	I*	1	Output	Number of times integrand is evaluated
10	isw	I	1	Input	Integer from 1 through 6, where isw = 1 is applicable for a peak-type function and isw = 6 is applicable for an oscillatory function. (See Notes (b))
11	iwk	I*	idv	Work	Work area (a size of 500 is reserved if zero is entered for idv)
12	wk	{D∗ / R∗}	4 × idv	Work	Work area (a size of 2000 is reserved if zero is entered for idv)
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) idv > 1 (except when 0 is input since the default value is assumed)

(e) 1 ≤ isw ≤ 6

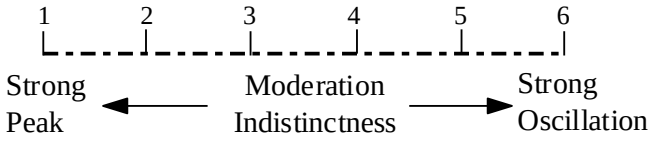
(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by −1 or the integral value= 0.0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv	Processing is terminated without obtaining a solution having the required precision.
2400	A certain subdivided interval cannot be further subdivided.	
2500	The solution precision will not reach the required precision.	
3000	Restriction (e) was not satisfied.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(b) The argument isw should be used according to the criteria shown in the following figure.



(c) This function uses an adaptive Gauss-Kronrod rule.

(7) **Example**

(a) Problem

Obtain the value of $\int_0^{0.9} \sin(10\pi x)dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

a=0.0, b=0.9, er=1.0e−10, ea=0.0, idv=0 and isw=6.

(c) Main program

```
/*      C interface example for ASL_dhnpnl */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return sin(10.0*M_PI*(*x));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double ers;
    double eas;
    int idvs;
    int idv0=500;
    double q;
    double ae;
    int nev;
    int isw;
    int *iwk;
    double *wk;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnpnl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnpnl ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &ers );
    fscanf( fp, "%lf", &eas );
    fscanf( fp, "%d", &idvs );
    fscanf( fp, "%d", &isw );

    iwk = ( int * )malloc((size_t)( sizeof(int) * idv0 ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (4*idv0) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\ters   = %8.3g\n", ers );
    printf( "\teas   = %8.3g\n", eas );
    printf( "\tidv   =   %6d\n", idvs );
    printf( "\tisw   =   %6d\n", isw );

    fclose( fp );

    ierr = ASL_dhnpnl(f, a, b, ers, eas, idvs, &q, &ae, &nev, isw, iwk, wk);

    printf( "\n     ** Output **\n\n" );
```

```
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tIntegral Approximation\n" );
        printf( "\t  q   = %8.3g\n", q );
        printf( "\n\tEstimate of Absolute Error\n" );
        printf( "\t  ae  = %8.3g\n", ae );
        printf( "\n\tNumber of Function Evaluations\n" );
        printf( "\t  nev = %6d\n", nev );

        free( iwk );
        free(  wk );

        return 0;
    }
```

(d) Output results

```
*** ASL_dhnpnl ***

 ** Input **

a     =        0
b     =      0.9
ers   =    1e-10
eas   =        0
idv   =        0
isw   =        6

 ** Output **

ierr =      0

Integral Approximation
  q   =   0.0637

Estimate of Absolute Error
  ae  = 2.56e-16

Number of Function Evaluations
  nev =     61
```

4.2.7 ASL_dhnenl, ASL_rhnenl
General Function Having an Endpoint Singularity

(1) **Function**

ASL_dhnenl or ASL_rhnenl integrates a general function having an endpoint singularity over a finite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnenl  (f, a, b, er, ea, itmx, &q, &ae, &nev, isw);
```

Single precision:

```
ierr = ASL_rhnenl  (f, a, b, er, ea, itmx, &q, &ae, &nev, isw);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error $\times 64$) (See Section 4.1.1)
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
6	itmx	I	1	Input	Maximum number of repetitions between singular points (Minimum subdivision width after DE transformation is $0.25 \times 2^{-\mathrm{itmx}}$; default value: 8)
7	q	{D∗ / R∗}	1	Output	Integral value
8	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
9	nev	I*	1	Output	Number of times integrand is evaluated
10	isw	I	1	Input	isw ≤ 0 : $f(x)$ is integrated directly. isw ≥ 1 : A canceling prevention transformation is applied to $f(x)$.
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) itmx > 1 (except when 0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by -1 or the integral value= 0.0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or itmx set to the default value.
2000	Number of times integrand is evaluated reached itmx	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3100	After DE transformation, the function value did not become smaller at both the + and − sides.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integration interval. (For example, you can set the function value at singular points to 0.0.)

(b) If the integrand $f(x)$ contains a factor of $(x-a)^{\alpha}$ or $(b-x)^{\beta}$ $(-1 < \alpha, \beta < 0)$, then you must perform the following kind transformation to prevent canceling.
First, rewrite the function f as a function of y where:

$a < x < \frac{a+b}{2}$ in places where $x = a - y(-\frac{b-a}{2} < y < 0)$
$b > x \geq \frac{a+b}{2}$ in places where, $x = b - y(0 < y \leq \frac{b-a}{2})$

and set isw = 1 and use the original interval (a, b).

For example, when calculating

$$\int_0^1 g(x)/\sqrt{x \cdot (1-x)dx}$$

this integral is expressed by:

$$\int_0^1 g(x)/\sqrt{x \cdot (1-x)}dx = \int_{-\frac{1}{2}}^0 g(-y)/\sqrt{(-y)(1+y)}dy + \int_0^{\frac{1}{2}} g(1-y)/\sqrt{(1-y)y}dy$$

so, input the following function `F` to this function. Further set isw = 1 and use *original* interval (0.0, 1.0).

```
/* C interface example for ASL_dhnenl */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include <asl.h>
double FORTRAN f(double *x)
{
    if(*x >= 0.0){/* Transformation to prevent canceling */
        return(g(1.0-(*x))/sqrt((*x)*(1.0 - (*x))));
    }else{
        return(g(-(*x))/sqrt(-(*x)*(1.0 + (*x))));
    }
}
double FORTRAN g(double *x)
{
    /* g(x) is an arbitrary function that specified by user. */
}
/* Main program */
int main()
{
......
    /* Call ASL function */
    ierr = ASL_dhnenl(f, a, b, epsr, epsa,...);
......
}
```

(c) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(d) This function uses a double exponential formula (DE transformation formula).

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

(f=$\begin{cases} 1/\sqrt{(2-\mathrm{x})\mathrm{x}} (0 < \mathrm{x} \leq 1) \\ 1/\sqrt{-\mathrm{x}(2+\mathrm{x})} (-1 < \mathrm{x} < 0) \end{cases}$.)

a=-1.0, b=1.0, er=0.0, ea=0.0, itmx=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dhnenl */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x >= 0.0 )
        return 1.0/sqrt((2.0-(*x))*(*x));
```

```
    else
        return 1.0/sqrt(-(*x)*(2.0+(*x)));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double epsr;
    double epsa;
    int lim;
    double result;
    double abserr;
    int neval;
    int isw;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnenl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnenl ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &epsr );
    fscanf( fp, "%lf", &epsa );
    fscanf( fp, "%d", &lim );
    fscanf( fp, "%d", &isw );

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\ter    = %8.3g\n", epsr );
    printf( "\tea    = %8.3g\n", epsa );
    printf( "\titmx  =   %6d\n", lim );
    printf( "\tisw   =   %6d\n", isw );

    fclose( fp );

    ierr = ASL_dhnenl(f, a, b, epsr, epsa, lim, &result, &abserr, &neval, isw);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", neval );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnenl ***

 ** Input **

a     =       -1
b     =        1
er    =        0
ea    =        0
itmx  =        0
isw   =        1

 ** Output **

ierr =      0

Integral Approximation
  q   =     3.14

Estimate of Absolute Error
  ae  = 5.58e-15

Number of Function Evaluations
  nev =     65
```

4.2.8 ASL_dhninl, ASL_rhninl
General Function Having Interior-Point Singularities

(1) **Function**

ASL_dhninl or ASL_rhninl integrates a general function having interior-point singularities over a finite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhninl  (f, a, b, sp, nsp, er, ea, itmx, &q, &ae, &nev);
```

Single precision:

```
ierr = ASL_rhninl  (f, a, b, sp, nsp, er, ea, itmx, &q, &ae, &nev);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	sp	{D∗ / R∗}	nsp	Input	Singular point X-coordinate values
				Output	Coordinate values sorted in ascending order
5	nsp	I	1	Input	Number of singular points
6	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
7	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
8	itmx	I	1	Input	Maximum number of repetitions between singular points (Minimum subdivision width after DE transformation is $0.25 \times 2^{-\mathrm{itmx}}$; default value: 8)
9	q	{D∗ / R∗}	1	Output	Integral value
10	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
11	nev	I*	1	Output	Number of times integrand is evaluated
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) itmx > 1 (except when 0 is input since the default value is assumed)

(e) nsp > 0

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by −1 or the integral value= 0.0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or itmx set to the default value.
2000	Number of times integrand is evaluated reached itmx	Processing is terminated without obtaining a solution having the required precision.
2300	The integration precision is bad in a certain interval.	
2500	The solution precision will not reach the required precision.	
3000	Restriction (e) was not satisfied.	Processing is aborted.
3100	After DE transformation, the function value did not become smaller at both the + and − sides.	
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integration interval. (For example, you can set the function value at singular points to 0.0.)

(b) If the singularity is severe at a singular point, the required precision may not be achieved, and only two digits will be obtained for single precision, only four digits for double precision format. Therefore, if you require higher precision, subdivide the interval at the singular points, perform a canceling prevention transformation, and integrate using the functions for functions having endpoint singularities 4.2.7 $\left\{\begin{matrix}\text{ASL\_dhnenl}\\ \text{ASL\_rhnenl}\end{matrix}\right\}$.

(c) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(d) This function obtains the integral value between each pair of singular points by using a double exponential formula (DE transformation formula) and then they calculate the total integral value by adding these together.

(7) **Example**

(a) Problem

Obtain the value of $\int_{-1}^{1} \frac{1}{\sqrt[3]{x^2}} dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

(Assume f=0.0 when $x = 0.0$.)

a=−1.0, b=1.0, sp[0]=0.0, nsp=1, er=1.0e−4, ea=0.0 and itmx=0.

(c) Main program

```
/*      C interface example for ASL_dhninl */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else if( *x > 0.0)
        return (pow((*x),(-2.0/3.0)));
    else
        return (pow(-(*x),(-2.0/3.0)));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double *point;
    int npts;
    double epsrel;
    double epsabs;
    int limit;
    double result;
    double abserr;
    int neval;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dhninl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhninl ***\n" );
    printf( "\n     ** Input **\n\n" );
    npts=1;

    point = ( double * )malloc((size_t)( sizeof(double) * npts ));
    if( point == NULL )
    {
        printf( "no enough memory for array point\n" );
        return -1;
    }
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    for( i=0 ; i<npts ; i++ )
```

```
        {
                fscanf( fp, "%lf", &point[i] );
        }
        fscanf( fp, "%lf", &epsrel );
        fscanf( fp, "%lf", &epsabs );
        fscanf( fp, "%d", &limit );

        printf( "\ta         = %8.3g\n", a );
        printf( "\tb         = %8.3g\n", b );
        for( i=0 ; i<npts ; i++ )
        {
                printf( "\tsp[%6d]= %8.3g\n", i,point[i] );
        }
        printf( "\ter        = %8.3g\n", epsrel );
        printf( "\tea        = %8.3g\n", epsabs );
        printf( "\titmx      =   %6d\n", limit );

        fclose( fp );

        ierr = ASL_dhninl(f, a, b, point, npts, epsrel, epsabs, limit, &result, &abserr, &neval);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tSorted X-Coordinate Value of The Singular Point\n" );
        for( i=0 ; i<npts ; i++ )
        {
                printf( "\t  sp[%6d]= %8.3g\n", i,point[i] );
        }
        printf( "\n\tIntegral Approximation\n" );
        printf( "\t  q   = %8.3g\n", result );
        printf( "\n\tEstimate of Absolute Error\n" );
        printf( "\t  ae  = %8.3g\n", abserr );
        printf( "\n\tNumber of Function Evaluations\n" );
        printf( "\t  nev = %6d\n", neval );

        free( point );

        return 0;
}
```

(d) Output results

```
*** ASL_dhninl ***

 ** Input **

a         =       -1
b         =        1
sp[     0]=        0
er        =   0.0001
ea        =        0
itmx      =        0

 ** Output **

ierr =      0

Sorted X-Coordinate Value of The Singular Point
  sp[     0]=        0

Integral Approximation
  q   =        6

Estimate of Absolute Error
  ae  = 3.16e-06

Number of Function Evaluations
  nev =     90
```

4.2.9 ASL_dhnanl, ASL_rhnanl
Singular Function for which Singularity Information is Unknown

(1) **Function**

ASL_dhnanl or ASL_rhnanl integrates a function having an endpoint or interior-point singularity for which singularity information is unknown over a finite interval. Calculations for these functions take a long time.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnanl  (f, a, b, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnanl  (f, a, b, er, ea, idv, &q, &ae, &nev, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	b	{D / R}	1	Input	Upper end of the integral interval
4	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
6	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value :500)
7	q	{D∗ / R∗}	1	Output	Integral value
8	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
9	nev	I*	1	Output	Number of times integrand is evaluated
10	iwk	I*	idv	Work	Work area (a size of 1000 should be reserved if zero is entered for idv)
11	wk	{D∗ / R∗}	4 × idv	Work	Work area (a size of 2000 should be reserved if zero is entered for idv)
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) er ≥ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) idv > 1 (except when 0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value over the interval (b, a) is multiplied by −1 or the integral value= 0.0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv	Processing is terminated without obtaining a solution having the required precision.
2400	A certain subdivided interval cannot be further subdivided.	
2500	The solution precision will not reach the required precision.	
2700	Solutions obtained according to the ε-algorithm did not converge.	
3500	The result is unreliable (the error is larger than the result).	Processing is aborted.

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integration interval. (For example, you can set the function value at singular points to 0.0).

(b) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(c) This function obtains solutions based on the 10-21 point adaptive Gauss-Kronrod rule. Convergence is accelerated in the vicinity of singular points where convergence is difficult to obtain by extrapolating according to the ε-algorithm.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_0^1 \frac{\log x}{\sqrt{x}} dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.
(Assume f=0.0 when $x = 0.0$.)
a=0.0, b=1.0, er=1.0e−8, ea=0.0 and idv=0.

(c) Main program

```
/*      C interface example for ASL_dhnanl */

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

#ifdef __cplusplus
extern "C"
{
#endif
```

```
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
    {
        return  (log(*x)/sqrt(*x));
    }
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double ers;
    double eas;
    int idvs;
    int idv0=500;
    double q;
    double ae;
    int nev;
    int *iwk;
    double *wk;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnanl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnanl ***\n" );
    printf( "\n     ** Input **\n\n" );
    idvs=0;
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &ers );
    fscanf( fp, "%lf", &eas );

    iwk = ( int * )malloc((size_t)( sizeof(int) * idv0 ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (4*idv0) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\ters   = %8.3g\n", ers );
    printf( "\teas   = %8.3g\n", eas );
    printf( "\tidv   =   %6d\n", idvs );

    fclose( fp );

    ierr = ASL_dhnanl(f, a, b, ers, eas, idvs, &q, &ae, &nev, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );

    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", q );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", ae );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev =   %6d\n", nev );

    free( iwk );
    free(  wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnanl ***

 ** Input **

a     =         0
b     =         1
ers   =     1e-08
eas   =         0
idv   =         0

 ** Output **

ierr =       0

Integral Approximation
  q    =        -4

Estimate of Absolute Error
  ae   = 4.25e-13

Number of Function Evaluations
  nev =       357
```

4.2.10 ASL_dhbdfs, ASL_rhbdfs
Integral of Product with any Function $f(x)$ and Bessel Function $J_0(x)$

(1) **Function**

Evaluate the integrals of the product with any function $f(x)$ and Bessel function $\int_0^1 J_0(\alpha_i x) f(x) x dx$ for positive parameters α_i.

(2) **Usage**

Double precision:

ierr = ASL_dhbdfs (m, ng, z, isw, f, b, work);

Single precision:

ierr = ASL_rhbdfs (m, ng, z, isw, f, b, work);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	m	I	1	Input	Number of parameter α_i
2	ng	I	1	Input	Number of data-points n
3	z	{D∗ / R∗}	m	Input	Positive parameter α_i
4	isw	I	1	Input	Integration method (See Notes (a))
5	f	–	–	Input	Name defining $f(x)$ (See Notes (b) and (c))
6	b	{D∗ / R∗}	m	Output	$\int_0^1 J_0(\alpha_i x) f(x) x dx$
7	work	{D∗ / R∗}	4×ng	Work	Work area
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) m $\geq$ 1

(b) ng $\geq$ 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) or (b) was not satisfied.	Processing is aborted.
4000	The sequence did not converge in the step where Gauss points were obtained.	
5000	Overflow occurred in evaluating of Legendre polynomials.	

(6) **Notes**

(a) If isw=0, then Gauss-ng points low is applied, but if isw≠0, evaluate $f(x)$ in $[0,1]$ at the $2 \times \mathrm{ng} - 1$ points to calculate analytically.

(b) f should be defined as follows.
double precision

```
void FORTRAN sfun(double *x, double *y)
{
      *y=f(*x);
}
```

single precision

```
void FORTRAN sfun(float *x,  float *y)
{
      *y=f(*x);
}
```

(c) $f(x)$ suffices to be defined in $0 \leq x \leq 1$.

(d) If z has an element α greater than 15.0, it is better to use 4.2.11 $\left\{\begin{matrix}\text{ASL\_dhbsfc}\\ \text{ASL\_rhbsfc}\end{matrix}\right\}$ than to use this function with setting isw=0 (namely applying Gauss formula).

(e) It is desirable to set ng to a large value to get a good precision. On the other hand, the calculation time will increase if the order n becomes large.

(f) If the value α is not known in advance, a practical precision can be obtained by setting isw=1 in almost cases.

(7) **Example**

(a) Problem
Set α=3.8352,4.1954,ng=50,isw=0(Gauss formula),

$$f(r) = 0.854 - 0.483r(2.845 + 1.726r(1.429 + 0.396r(0.472 - 3.172r(1.741 - 2.621r$$

$$(2.459 - 1.637r(1.28 + 3.284r))))))$$

to evaluate $\int_0^1 J_0(\alpha x) f(x) x dx$, and compare the result with isw=1.

(b) Main program

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

#ifdef __cplusplus
extern "C"
{
```

```
#endif
#ifdef __STDC__
void xtoy(double *x, double *y)
#else
void xtoy(x, y)
double *x;
double *y;
#endif
{
    *y=1.28 +3.284*(*x);
    *y=2.459-1.637*(*x)*(*y);
    *y=1.741-2.621*(*x)*(*y);
    *y=0.472-3.172*(*x)*(*y);
    *y=1.429+0.396*(*x)*(*y);
    *y=2.845+1.726*(*x)*(*y);
    *y=0.854-0.483*(*x)*(*y);
}
#ifdef __cplusplus
}
#endif
int main()
{
    double *aa, *bb, *work;
    int    m,ng,isw,ierr;
    m=2;ng=50;isw=0;
    aa=(double * )malloc((size_t)( sizeof(double)* m ));
    if( aa == NULL )
    {
        printf( "no enough memory for array aa \n" );
        return -1;
    }
    bb=(double * )malloc((size_t)( sizeof(double)* (m*2) ));
    if( bb == NULL )
    {
        printf( "no enough memory for array bb  \n" );
        return -1;
    }
    work=(double * )malloc((size_t)( sizeof(double)* (4*ng) ));
    if( work == NULL )
    {
        printf( "no enough memory for array work  \n" );
        return -1;
    }
    aa[0]=3.8352;
    aa[1]=4.1954;
    printf( "\n\t *** ASL_dhbdfs   \n\n" );
    printf( "\n\t *** input    aa \n\n" );
    printf( "\n%13.8g , %13.8g\n", aa[0],aa[1]);
    ierr=ASL_dhbdfs(m, ng, aa, isw, xtoy, bb, work);
    printf( "\n\t *** OUTPUT ***\n\n" );
    printf( "\n\tierr = %6d\n", ierr );
    printf( "\n\t  bb[0],bb[1]  \n\n" );
    printf( "\n%13.8g , %13.8g\n",bb[0],bb[1]);
    isw=1;
    ierr=ASL_dhbdfs(m, ng, aa, isw, xtoy, &bb[2], work);
    printf( "\n\tierr = %6d\n", ierr );
    printf( "\n%13.8g , %13.8g\n",bb[2],bb[3]);
    printf( "\n%13.8g , %13.8g\n",bb[2]-bb[0],bb[3]-bb[1]);
    free(aa);
    free(bb);
    free(work);
    return 0;
}
```

(c) Output results

```
 *** ASL_dhbdfs

 *** input    aa

   3.8352 ,        4.1954
 *** OUTPUT ***

ierr =      0

  bb[0],bb[1]

-0.45758033 ,   -0.48041803

ierr =      0

-0.45758082 ,    -0.4804185

-4.9076906e-07 , -4.7077319e-07
```

4.2.11 ASL_dhbsfc, ASL_rhbsfc
Integral of the Product of Chebyshev Polynomial and Bessel Function of the Order 0

(1) **Function**

Evaluate the integral of the product of Chebyshev polynomial and Bessel function of the order 0 $\int_0^1 J_0(\alpha_i x)T_n(2x-1)xdx$ for $i = 1, \cdots$ and $n = 0, \cdots$.

(2) **Usage**

Double precision:

```
ierr = ASL_dhbsfc  ( n, m, z, c, nc, work);
```

Single precision:

```
ierr = ASL_rhbsfc  ( n, m, z, c, nc, work);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	n	I	1	Input	Upper bound of degree of Chebyshev polynomials n
2	m	I	1	Input	Number of parameters α_i
3	z	{D∗ / R∗}	m	Input	Parameters α_i
4	c	{D / R}	See Contents	Output	$\int_0^1 J_0(\alpha_i x)T_l(2x-1)xdx (l = 0, \cdots,$ n) **Size**: (nc + 1) × m
5	nc	I	1	Input	Adjustable dimension of array c
6	work	{D∗ / R∗}	(n+1)×3	Work	Work area
7	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) 1 ≤ n ≤ nc

(b) m ≥ 1

(c) z is positive.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a), (b) or (c) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) In the case where z has some elements less than 15.0, it is better for these values to apply 4.2.10 $\left\{\begin{matrix}\text{ASL\_dhbdfs}\\ \text{ASL\_rhbdfs}\end{matrix}\right\}$.

(b) n should be less than 36.

(c) $\int_0^1 J_0(\alpha x)T_l(2x-1)xdx(l=0,\cdots,\text{n})$ is output to c[l+(n+1)*j](α=z[j]).

(7) **Example**

(a) Problem

Set α=3.8352,4.1954,n=7 to evaluate $\int_0^1 J_0(\alpha x)T_l(2x-1)xdx$. By this result and the Fourier coefficients of the function

$$f(r)=0.854-0.483r(2.845+1.726r(1.429+0.396r(0.472-3.172r(1.741-2.621r$$

$$(2.459-1.637r(1.28+3.284r))))))$$

get the value of $\int_0^1 J_0(\alpha x)f(x)xdx$.

(b) Main program

```
#include <stdio.h>
#include <stdlib.h>
#include <asl.h>
#include <math.h>
int main()
{
    int n,m,ng;
    int i,j,k,ierr;
    double *aa, *cf, *work, *fr, *b, *fn;
    double x,y;
    n=7;m=2;ng=50;
    aa=(double * )malloc((size_t)( sizeof(double)* m ));
    if( aa == NULL )
    {
        printf( "no enough memory for array aa\n" );
        return -1;
    }
    aa[0]=3.8352;
    aa[1]=4.1954;
    cf=(double * )malloc((size_t)( sizeof(double)* (m*(1+n)) ));
    if( cf == NULL )
    {
        printf( "no enough memory for array cf\n" );
        return -1;
    }
    work=(double * )malloc((size_t)( sizeof(double)* (3*(1+n)) ));
    if( work == NULL )
    {
        printf( "no enough memory for array work\n" );
        return -1;
    }
    fr=(double * )malloc((size_t)( sizeof(double)* (1+n) ));
    if( fr == NULL )
    {
        printf( "no enough memory for array fr \n" );
        return -1;
    }
    b=(double * )malloc((size_t)( sizeof(double)* m ));
    if(b  == NULL )
    {
        printf( "no enough memory for array b\n" );
        return -1;
    }
    fn=(double * )malloc((size_t)( sizeof(double)* ng ));
    if( fn == NULL )
    {
        printf( "no enough memory for array fn\n" );
        return -1;
    }
    printf( "\n\t *** ASL_dhbsfc \n\n" );
    printf( "\n\t input \n\n" );
    printf( "\n %13.8g,%13.8g\n",aa[0],aa[1]);
    ierr=ASL_dhbsfc(n, m, aa, cf, n, work);
    printf( "\n\t *** OUTPUT ***\n\n" );
    printf( "\n\tierr = %6d\n", ierr );
    printf( "\n\ti          cf1          cf2  \n\n" );
```

```
    for ( i=0 ; i<n+1 ; i++ )
    {
        printf( "\n%9d,%13.8g,%13.8g\n",i,cf[i],cf[i+n+1]);
    }
/*    fourier transform */
    for ( k=0 ; k<ng ; k++ )
    {
        x=(cos((M_PI*(2*k+1))/(2*ng))+1.0)/2.0;
        y=1.28 +3.284*x;
        y=2.459-1.637*x*y;
        y=1.741-2.621*x*y;
        y=0.472-3.172*x*y;
        y=1.429+0.396*x*y;
        y=2.845+1.726*x*y;
        y=0.854-0.483*x*y;
        fn[k]=y;
    }
    for ( i=0 ; i<n+1 ; i++ )
    {
        fr[i]=0.0;
        for ( k=0 ; k<ng ; k++ )
        {
            fr[i]=fr[i]+fn[k]*cos((i*M_PI*(2*k+1))/(2*ng));
        }
        fr[i]=fr[i]*2.0/ng;
        if(i == 0)
        {
            fr[i]=fr[i]/2.0;
        }
    }
    for ( i=0 ; i<m ; i++ )
    {
        b[i]=0.0;
        for ( j=0 ; j<n+1 ; j++ )
        {
            b[i]=b[i]+fr[j]*cf[j+(n+1)*i];
        }
    }
    printf( "\n\t coefficient \n\n" );
    printf( "\n %13.8g,%13.8g\n",b[0],b[1]);
    free(cf);
    free(work);
    free(fr);
    free(b);
    free(fn);
    free(aa);
    return 0;
}
```

(c) Output results

```
 *** ASL_dhbsfc

 input

    3.8352,       4.1954
 *** OUTPUT ***

ierr =      0
i          cf1          cf2

    0,-0.00036676299, -0.032670159
    1, -0.093804025,  -0.10122805
    2, -0.063933641, -0.041917162
    3,  0.075850369,  0.089379895
    4,  0.044986007,  0.041434474
    5,  0.001469729,-0.0034078357
    6, 0.0042640297, 0.0030696751
    7, 0.0040831492, 0.0038106815
 coefficient

-0.45758033,  -0.48041803
```

4.3 INTEGRATION OVER A SEMI-INFINITE INTERVAL

4.3.1 ASL_dhemnh, ASL_rhemnh
Arbitrary Function

(1) **Function**

ASL_dhemnh or ASL_rhemnh automatically integrates the function over a semi-infinite interval. Even if the integrand has singularities, ASL_dhemnh or ASL_rhemnh determines its characteristics and automatically processes it. These functions are easy to use since the number of required input arguments has been minimized.

(2) **Usage**

Double precision:

```
ierr = ASL_dhemnh  (f, a, er, &q, &ae);
```

Single precision:

```
ierr = ASL_rhemnh  (f, a, er, &q, &ae);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
4	q	{D∗ / R∗}	1	Output	Integral value
5	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
6	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) er $\geq$ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	There are five or more singular points.	Processing continues.
1500	Restriction (a) was not satisfied.	Processing is performed with er set to the default value.
2000	Number of times integrand is evaluated reached 500	The minimum subdivision width is gradually widened so that an approximate solution can be obtained.
2400	A certain subdivided interval cannot be further subdivided.	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3500	The result is unreliable (the error is larger than the result).	Processing is aborted.
4000	Overflow occurred more than once in a single subdivided interval.	

(6) **Notes**

(a) When the integrand has singularities, if the required precision is not more lenient than the default value, the solution precision may worsen and the output for the number of singular points may be greater than the actual number.
If the integrand has numerous peaks, you should increase the required precision by using the double-precision function when obtaining the solution.
If the integrand is oscillatory, you should use the function 4.3.2 $\left\{\begin{matrix}\text{ASL\_dhnofh}\\ \text{ASL\_rhnofh}\end{matrix}\right\}$.
At other times or if you have no information about function characteristics, you should specify the precision you require as the required precision when obtaining the solution.

(b) If you input 0.0 for the variable er, the default value is set.

(c) This function uses an algorithm that is based on the adaptive Newton-Cotes 9-point rule but having more powerful singular point processing capabilities.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_2^\infty \frac{1}{x^2}dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

a=2.0 and er=0.0.

(c) Main program

```
/*      C interface example for ASL_dhemnh */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return 1.0/((*x) * (*x));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double epsrel;
    double result;
    double abserr;
    int ierr;
    FILE *fp;

    fp = fopen( "dhemnh.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhemnh ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &epsrel );

    printf( "\ta   = %8.3g\n", a );
    printf( "\ter  = %8.3g\n", epsrel );

    fclose( fp );

    ierr = ASL_dhemnh(f, a, epsrel, &result, &abserr);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );

    return 0;
}
```

(d) Output results

```
*** ASL_dhemnh ***

 ** Input **

a   =        2
er  =        0

 ** Output **

ierr =      0
```

```
Integral Approximation
  q   =      0.5

Estimate of Absolute Error
  ae  = 8.88e-16
```

4.3.2 ASL_dhnofh, ASL_rhnofh
Function of the Type $f(x)\cdot$ ($\sin\omega x$ or $\cos\omega x$)

(1) **Function**

ASL_dhnofh or ASL_rhnofh integrates an oscillatory function that can be factored into $f(x)\cdot$ ($\sin\omega x$ or $\cos\omega x$) over a semi-infinite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnofh  (f, a, w, itype, ea, isy, idv, &q, &ae, &nev, &iwk, &wk);
```

Single precision:

```
ierr = ASL_rhnofh  (f, a, w, itype, ea, isy, idv, &q, &ae, &nev, &iwk, &wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the factor of integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	w	{D / R}	1	Input	ω of the weight function ($\sin\omega x$ or $\cos\omega x$)
4	itype	I	1	Input	Weight function type $1\cdots\int f(x)\cos(\omega x)dx$ $2\cdots\int f(x)\sin(\omega x)dx$
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
6	isy	I	1	Input	Maximum number of repetitions between singular points (Minimum subdivision width after de transformation is $0.25\times 2^{-\mathrm{isy}}$; default value: 8)
7	idv	I	1	–	Unused
8	q	{D∗ / R∗}	1	Output	Integral value
9	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
10	nev	I*	1	Output	Number of times integrand is evaluated
11	iwk	I*	1	Work	Work area (unused) ; passing dummy pointer.
12	wk	{D∗ / R∗}	1	Work	Work area (unused) ; passing dummy pointer.
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) ea $\geq$ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(b) $2 <$ isy < 51 (except when 0 is input since the default value is assumed)

(c) itype =1 or 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) or (b) was not satisfied.	Processing is performed with ea, idv, isy set to the default value.
2100	The number of repetitions reached isy	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3000	Restriction (c) was not satisfied.	Processing is aborted.
3100	After DE transformation, the function value did not become smaller at both the + and − sides.	
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument. 0.0 is input.

(b) This function calculates with the variable conversion type formula for an oscillatory function semi-infinite integration.

(c) If the required absolute precision is not reached, then the solution is returned by making the convergence decision using 64× (Unit for determining error) as the relative precision.

(7) **Example**

(a) Problem
Obtain the value of $\displaystyle\int_0^\infty \frac{\sin x}{x}dx$.

(b) Input data
Function name corresponding to integrand $f(x)$: f.
(Assume f=0.0 when $x = 0.0$.)
a=0.0, w=1.0, itype=2, ea=1.0e−8 and isy=0.

(c) Main program

```
/*      C interface example for ASL_dhnofh */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
        return 1.0/(*x);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double om;
    int ity;
    double epsab;
    int lims;
    int lit=0; /* dummy */
    double s;
    double abser;
    int neval;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnofh.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnofh ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &om );
    fscanf( fp, "%d", &ity );
    fscanf( fp, "%lf", &epsab );
    fscanf( fp, "%d", &lims );

    printf( "\ta     = %8.3g\n", a );
    printf( "\tw     = %8.3g\n", om );
    printf( "\titype =   %6d\n", ity );
    printf( "\tea    = %8.3g\n", epsab );
    printf( "\tisy   =   %6d\n", lims );

    fclose( fp );

    ierr = ASL_dhnofh
    (f, a, om, ity, epsab, lims, lit, &s, &abser, &neval, NULL, NULL);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", s );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abser );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", neval );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnofh ***

 ** Input **

a     =        0
w     =        1
itype =        2
```

```
ea    =    1e-08
isy   =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =     1.57

Estimate of Absolute Error
  ae  = 5.69e-10

Number of Function Evaluations
  nev =     73
```

4.3.3 ASL_dhnenh, ASL_rhnenh
General Function Having an Endpoint Singularity

(1) **Function**

ASL_dhnenh or ASL_rhnenh integrates an general function having an endpoint singularity over a semi-infinite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnenh  (f, a, er, ea, itmx, &q, &ae, &nev, isw);
```

Single precision:

```
ierr = ASL_rhnenh  (f, a, er, ea, itmx, &q, &ae, &nev, isw);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
4	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
5	itmx	I	1	Input	Maximum number of repetitions between singular points (Minimum subdivision width after de transformation is $0.25 \times 2^{-\mathrm{itmx}}$; default value: 8)
6	q	{D∗ / R∗}	1	Output	Integral value
7	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
8	nev	I*	1	Output	Number of times integrand is evaluated

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
9	isw	I	1	Input	≤ 0: The integrand does not have a factor of e^{-x} or it is unknown if it has such a factor ≥ 1: The integrand has a factor of e^{-x}
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) er > Unit for determining error $\times 64$
(except when 0.0 is input since the default value is assumed)

(b) ea $\geq$ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(c) itmx > 1 (except when 0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a), (b) or (c) was not satisfied.	Processing is performed with er, ea or itmx set to the default value.
2000	Processing was repeated itmx times.	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3100	After DE transformation, the function value did not become smaller sufficiently at both the + and − sides.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(b) This function uses a double exponential formula (DE transformation formula).

(7) **Example**

(a) Problem

Obtain the value of $\int_0^\infty e^{-x}\log(x)dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

(Assume f=0.0 when $x = 0.0$.)

a=0.0, er=1.0e−8, ea=0.0, itmx=0 and isw=1.

(c) Main program

```
/*      C interface example for ASL_dhnenh */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
        return exp(-(*x))*log(*x);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double epsr;
    double epsa;
    int lim;
    double result;
    double abserr;
    int neval;
    int isw;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnenh.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnenh ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &epsr );
    fscanf( fp, "%lf", &epsa );
    fscanf( fp, "%d", &lim );
    fscanf( fp, "%d", &isw );

    printf( "\ta     = %8.3g\n", a );
    printf( "\ter    = %8.3g\n", epsr );
    printf( "\tea    = %8.3g\n", epsa );
    printf( "\titmx  =   %6d\n", lim );
    printf( "\tisw   =   %6d\n", isw );

    fclose( fp );

    ierr = ASL_dhnenh(f, a, epsr, epsa, lim, &result, &abserr, &neval, isw);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );
```

```
        printf( "\n\tNumber of Function Evaluations\n" );
        printf( "\t  nev = %6d\n", neval );

        return 0;
    }
```

(d) Output results

```
*** ASL_dhnenh ***

 ** Input **

a     =        0
er    =    1e-08
ea    =        0
itmx  =        0
isw   =        1

 ** Output **

ierr =      0

Integral Approximation
  q   =   -0.577

Estimate of Absolute Error
  ae  = 2.49e-12

Number of Function Evaluations
  nev =     81
```

4.3.4 ASL_dhninh, ASL_rhninh
General Function Having Interior-Point Singularities

(1) **Function**

ASL_dhninh or ASL_rhninh integrates a general function having interior-point singularities over a semi-infinite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhninh  (f, a, sp, nsp, er, ea, itmx, &q, &ae, &nev);
```

Single precision:

```
ierr = ASL_rhninh  (f, a, sp, nsp, er, ea, itmx, &q, &ae, &nev);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	a	{D / R}	1	Input	Lower end of the integral interval
3	sp	{D∗ / R∗}	nsp	Input	Singular point X-coordinate values
				Output	Coordinate values sorted in ascending order
4	nsp	I	1	Input	Number of singular points
5	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
6	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
7	itmx	I	1	Input	Maximum number of repetitions between singular points (Minimum subdivision width after de transformation is $0.25 \times 2^{-\mathrm{itmx}}$; default value: 8)
8	q	{D∗ / R∗}	1	Output	Integral value

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
9	ae	{D∗ R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
10	nev	I*	1	Output	Number of times integrand is evaluated
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) er $\geq$ Unit for determining error $\times 64$
(except when 0.0 is input since the default value is assumed)

(b) ea $\geq$ Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(c) itmx > 1 (except when 0 is input since the default value is assumed)

(d) nsp > 0

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a), (b) or (c) was not satisfied.	Processing is performed with er, ea or itmx set to the default value.
2000	Processing was repeated itmx times in one interval.	Processing is terminated without obtaining a solution having the required precision.
2300	The integral precision is bad in a certain interval.	
2500	The solution precision will not reach the required precision.	
3000	Restriction (d) was not satisfied.	Processing is aborted.
3100	After DE transformation, the function value did not become smaller at both the + and − sides.	
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integral interval. (For example, you can set the function value at singular points to 0.0.)

(b) If the singularity is severe at a singular point, the required precision may not be achieved, and only two digits will be obtained for single precision, only four digits for double precision. Therefore, if you require higher precision, subdivide the interval at the singular points, perform a canceling prevention transformation for each finite interval, and integrate using the functions for functions having endpoint singularities 4.2.7 {ASL_dhnenl ASL_rhnenl}, and then for the semi-infinite interval, integrate using the function

for functions having endpoint singularities 4.3.3 $\left\{\begin{matrix}\text{ASL\_dhnenh}\\ \text{ASL\_rhnenh}\end{matrix}\right\}$.

(c) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(d) This function obtains the integral value between each pair of singular points by using a double exponential formula (DE transformation formula) for a semi-infinite interval and various finite intervals and then they calculate the total integral value by adding these together.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_{-1}^{\infty} \frac{x}{e^x - 1} dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

(Assume f=0.0 when $x = 0.0$.)

a=−1.0, sp[0]=0.0, nsp=1, er=1.0e−3, ea=0.0 and itmx=0.

(c) Main program

```
/*      C interface example for ASL_dhninh */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
        return (*x)/( expm1(*x) );
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double *point;
    int npts;
    double epsrel;
    double epsabs;
    int limit;
    double result;
    double abserr;
    int neval;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dhninh.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhninh ***\n" );
    printf( "\n     ** Input **\n\n" );
    npts=1;

    point = ( double * )malloc((size_t)( sizeof(double) * npts ));
    if( point == NULL )
    {
        printf( "no enough memory for array point\n" );
        return -1;
```

```
        }
        fscanf( fp, "%lf", &a );
        for( i=0 ; i<npts ; i++ )
        {
                fscanf( fp, "%lf", &point[i] );
        }
        fscanf( fp, "%lf", &epsrel );
        fscanf( fp, "%lf", &epsabs );
        fscanf( fp, "%d", &limit );

        printf( "\ta         = %8.3g\n", a );
        for( i=0 ; i<npts ; i++ )
        {
                printf( "\tsp[%6d]= %8.3g\n", i,point[i] );
        }
        printf( "\ter        = %8.3g\n", epsrel );
        printf( "\tea        = %8.3g\n", epsabs );
        printf( "\titmx      =   %6d\n", limit );

        fclose( fp );

        ierr = ASL_dhninh(f, a, point, npts, epsrel, epsabs, limit, &result, &abserr, &neval);

        printf( "\n     ** Output **\n\n" );
        printf( "\tierr = %6d\n", ierr );
        printf( "\n\tSorted X-Coordinate Value of The Singular Point\n" );
        for( i=0 ; i<npts ; i++ )
        {
                printf( "\t  sp[%6d]= %8.3g\n", i,point[i] );
        }
        printf( "\n\tIntegral Approximation\n" );
        printf( "\t  q   = %8.3g\n", result );
        printf( "\n\tEstimate of Absolute Error\n" );
        printf( "\t  ae  = %8.3g\n", abserr );
        printf( "\n\tNumber of Function Evaluations\n" );
        printf( "\t  nev = %6d\n", neval );

        free( point );

        return 0;
}
```

(d) Output results

```
*** ASL_dhninh ***

 ** Input **

a         =       -1
sp[     0]=        0
er        =   0.0001
ea        =        0
itmx      =        0

 ** Output **

ierr =      0

Sorted X-Coordinate Value of The Singular Point
  sp[     0]=        0

Integral Approximation
  q   =     2.92

Estimate of Absolute Error
  ae  = 3.31e-05

Number of Function Evaluations
  nev =     78
```

4.4 INTEGRATION OVER A FULLY INFINITE INTERVAL

4.4.1 ASL_dhemni, ASL_rhemni
Arbitrary Function

(1) **Function**

ASL_dhemni or ASL_rhemni automatically integrates the function over the fully infinite interval from $-\infty$ to ∞. The integrand can even have singularities at internal points. These functions are easy to use since the number of required input/output arguments has been minimized.

(2) **Usage**

Double precision:

```
ierr = ASL_dhemni  (f, er, &q, &ae);
```

Single precision:

```
ierr = ASL_rhemni  (f, er, &q, &ae);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
3	q	{D∗ / R∗}	1	Output	Integral value
4	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
5	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) er $\geq$ Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) was not satisfied.	Processing is performed with er set to the default value.
2000	Number of times integrand is evaluated reached 500.	The minimum subdivision width is gradually widened so that an approximate solution can be obtained.
2400	A certain subdivided interval cannot further subdivided.	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3500	The result is unreliable (the error is larger than the result).	Processing is aborted.
4000	Overflow occurred more than once in a single subdivided interval.	

(6) **Notes**

(a) When the integrand has singularities, if the required precision is not more lenient than the default value, the solution precision may worsen and the output for the number of singular points may be greater than the actual number.
If the integrand has numerous peaks, you should increase the required precision by using the double-precision function when obtaining the solution.
If the integrand is oscillatory, you should use the function 4.4.2 $\left\{\begin{matrix}\text{ASL\_dhnofi}\\ \text{ASL\_rhnofi}\end{matrix}\right\}$.
At other times or if you have no information about function characteristics, you should specify the precision you require as the required precision when obtaining the solution.

(b) If you input 0.0 for the variable er, the default value is set.

(c) This function integrates over the fully infinite interval according to a variable transformation, using an algorithm that is based on the adaptive Newton-Cotes 9-point rule but having more powerful singular point processing capabilities.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_{-\infty}^{\infty} \frac{1}{1+x^2} dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.
er=0.0.

(c) Main program

```
/*      C interface example for ASL_dhemni */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return 1.0/(1.0 + (*x) * (*x));
}
#ifdef __cplusplus
}
#endif


int main()
{
    double epsr;
    double result;
    double abserr;
    int ierr;
    FILE *fp;

    fp = fopen( "dhemni.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhemni ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &epsr );

    printf( "\ter  = %8.3g\n", epsr );

    fclose( fp );

    ierr = ASL_dhemni(f, epsr, &result, &abserr);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );

    return 0;
}
```

(d) Output results

```
*** ASL_dhemni ***

 ** Input **

er  =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =     3.14
```

```
Estimate of Absolute Error
  ae  = 5.58e-15
```

4.4.2 ASL_dhnofi, ASL_rhnofi
Function of the Type $f(x) \cdot (\sin \omega x$ or $\cos \omega x)$

(1) **Function**

ASL_dhnofi or ASL_rhnofi integrates an oscillatory function that can be factored into $f(x) \cdot (\sin \omega x$ or $\cos \omega x)$ over the fully infinite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnofi  (f, w, itype, ea, isy, idv, &q, &ae, &nev, iwk, &wk);
```

Single precision:

```
ierr = ASL_rhnofi  (f, w, itype, ea, isy, idv, &q, &ae, &nev, iwk, &wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the factor of integrand $f(x)$
2	w	{D / R}	1	Input	ω of the weight function ($\sin \omega x$ or $\cos \omega x$)
3	itype	I	1	Input	Weight function type $1 \cdots \int f(x) \cos(\omega x) dx$ $2 \cdots \int f(x) \sin(\omega x) dx$
4	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum error $\times 2^{24}$) (See Section 4.1.1)
5	isy	I	1	Input	Maximum number of repetitions between singular points (Minimum subdivision width after de transformation is $0.25 \times 2^{-\mathrm{isy}}$; default value: 8)
6	idv	I	1	–	Unused
7	q	{D∗ / R∗}	1	Output	Integral value
8	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
9	nev	I*	1	Output	Number of times integrand is evaluated.
10	iwk	I*	1	Work	Work area (unused) ; passing dummy pointer.

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
11	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	1	Work	Work area (unused) ; passing dummy pointer.
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) ea $\geq$Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(b) $2 <$ isy < 51
(except when 0 is input since the default value is assumed)

(c) itype =1 or 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) or (b) was not satisfied.	Processing is performed with ea, or idv set to the default value.
2100	The number of repetitions in the semi-infinite interval on one side reached isy.	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the required precision.	
3000	Restriction (c) was not satisfied.	Processing is aborted.
3100	After DE transformation, the function value did not become smaller at both the + and − sides.	
3500	The result is unreliable (the error is larger than the result.)	

(6) **Notes**

(a) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(b) This function obtains the integral values over the intervals $(-\infty, 0]$ and $[0, \infty)$ based on the variable conversion formula for semi-infinite integration of an oscillatory function.

(c) If the required absolute precision is not reached, then the solution is returned by making the convergence decision using 64× (Unit for determining error) as the relative precision.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_{-\infty}^{\infty} \frac{\sin x}{x} dx$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

(Assume f=0.0 when $x = 0.0$.)

w=1.0, itype=2, ea=1.0e−8 and isy=0.

(c) Main program

```
/*      C interface example for ASL_dhnofi */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( *x == 0.0 )
        return 0.0;
    else
        return 1.0/(*x);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double omega;
    int integr;
    double epsabs;
    int limst;
    int limit=0;  /* dummy */
    double result;
    double abserr;
    int neval;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnofi.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnofi ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &omega );
    fscanf( fp, "%d", &integr );
    fscanf( fp, "%lf", &epsabs );
    fscanf( fp, "%d", &limst );

    printf( "\tw     = %8.3g\n", omega );
    printf( "\titype =   %6d\n", integr );
    printf( "\tea    = %8.3g\n", epsabs );
    printf( "\tisy   =   %6d\n", limst );

    fclose( fp );

    ierr = ASL_dhnofi
    (f, omega, integr, epsabs, limst, limit,
        &result, &abserr, &neval, NULL, NULL);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );
```

```
        printf( "\n\tNumber of Function Evaluations\n" );
        printf( "\t  nev = %6d\n", neval );


        return 0;
    }
```

(d) Output results

```
        *** ASL_dhnofi ***

         ** Input **

        w     =        1
        itype =        2
        ea    =    1e-08
        isy   =        0

         ** Output **

        ierr =      0

        Integral Approximation
          q   =     3.14

        Estimate of Absolute Error
          ae  = 3.29e-10

        Number of Function Evaluations
          nev =    156
```

4.4.3 ASL_dhnini, ASL_rhnini
Function Having Interior-Point Singularities

(1) **Function**

ASL_dhnini or ASL_rhnini integrates a singular function over the fully infinite interval.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnini  (f, sp, nsp, er, ea, itmx, &q, &ae, &nev);
```

Single precision:

```
ierr = ASL_rhnini  (f, sp, nsp, er, ea, itmx, &q, &ae, &nev);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x)$
2	sp	{D∗ / R∗}	nsp	Input	Singular point X-coordinate values
				Output	Coordinate values sorted in ascending order
3	nsp	I	1	Input	Number of singular points
4	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
6	itmx	I	1	Input	Maximum number of repetitions between singular points in one interval (Minimum subdivision width after de transformation is $0.25 \times 2^{-\mathrm{itmx}}$; default value: 8)
7	q	{D∗ / R∗}	1	Output	Integral value

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	ae	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	1	Output	Estimated value of absolute error (See Section 4.1.1)
9	nev	I*	1	Output	Number of times integrand is evaluated
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) er ≥Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(b) ea ≥Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(c) itmx > 1 (except when 0 is input since the default value is assumed)

(d) nsp > 0

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a), (b) or (c) was not satisfied.	Processing is performed with er, ea or itmx set to the default value.
2000	A single interval was subdivided itmx times.	Processing is terminated without obtaining a solution having the required precision.
2300	The integral precision is bad in a certain interval.	
2400	A certain subdivided interval cannot be further subdivided	
2500	The solution precision will not reach the required precision	
3000	Restriction (d) was not satisfied.	Processing is aborted.
3100	After DE transformation, the function value did not become smaller at both the + and − sides.	
3500	The result is unreliable (the error is larger than the result).	

(6) **Notes**

(a) You must make sure that a function value overflow does not occur within the integration interval. (For example, you can set the function value at singular points to 0.0.)

(b) If the singularity is severe at a singular point, the required precision may not be achieved, and only two digits will be obtained for single precision, or only four digits for double precision.
Therefore, if you require higher precision, subdivide the interval at the singular points, perform a can-

celing prevention transformation for each finite interval, and integrate using the functions for functions having endpoint singularities 4.2.7 $\left\{\begin{matrix}\text{ASL\_dhnenl}\\ \text{ASL\_rhnenl}\end{matrix}\right\}$.

(c) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(d) This function obtains the integral value between each pair of singular points by using a double exponential formula (DE transformation formula) for semi-infinite intervals and various finite intervals and then they calculate the total integral value by adding these together.

(7) **Example**

(a) Problem

Obtain the value of $\displaystyle\int_{-\infty}^{\infty} f(x)dx$ for $f(x) = \begin{cases} \frac{1}{x^2} & (|x| > 2) \\ x+2 & (|x| \leq 2) \end{cases}$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.
sp[0]=−2.0, sp[1]=2.0, nsp=2, er=1.0e−8, ea=0.0 and itmx=0.

(c) Main program

```
/*      C interface example for ASL_dhnini */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    if( fabs(*x) > 2.0 )
        return (1.0/( (*x) * (*x) ));
    else
        return ((*x)+2.0) ;
}
#ifdef __cplusplus
}
#endif

int main()
{
    double *point;
    int npts;
    double epsrel;
    double epsabs;
    int limit;
    double result;
    double abserr;
    int neval;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dhnini.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnini ***\n" );
    printf( "\n     ** Input **\n\n" );
    npts=2;

    point = ( double * )malloc((size_t)( sizeof(double) * npts ));
    if( point == NULL )
    {
        printf( "no enough memory for array point\n" );
```

```
        return -1;
    }
    for( i=0 ; i<npts ; i++ )
    {
            fscanf( fp, "%lf", &point[i] );
    }
    fscanf( fp, "%lf", &epsrel );
    fscanf( fp, "%lf", &epsabs );
    fscanf( fp, "%d", &limit );

    for( i=0 ; i<npts ; i++ )
    {
            printf( "\tsp[%6d]= %8.3g\n", i,point[i] );
    }
    printf( "\ter        = %8.3g\n", epsrel );
    printf( "\tea        = %8.3g\n", epsabs );
    printf( "\titmx      =   %6d\n", limit );

    fclose( fp );

    ierr = ASL_dhnini(f, point, npts, epsrel, epsabs, limit, &result, &abserr, &neval);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tSorted X-Coordinate Value of The Singular Point\n\n" );
    for( i=0 ; i<npts ; i++ )
    {
            printf( "\t  sp[%6d]= %8.3g\n", i,point[i] );
    }
    printf( "\n\tIntegral Approximation\n\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n\n" );
    printf( "\t  ae  = %8.3g\n", abserr );
    printf( "\n\tNumber of Function Evaluations\n\n" );
    printf( "\t  nev = %6d\n", neval );

    free( point );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnini ***

 ** Input **

sp[     0]=       -2
sp[     1]=        2
er        =        0
ea        =        0
itmx      =        0

 ** Output **

ierr =      0

Sorted X-Coordinate Value of The Singular Point

  sp[     0]=       -2
  sp[     1]=        2

Integral Approximation

  q   =        9

Estimate of Absolute Error

  ae  = 3.02e-14

Number of Function Evaluations

  nev =    350
```

4.4.4 ASL_dh2int, ASL_rh2int
Function of the Type $e^{-x^2} \cdot f(x)$

(1) **Function**

Evaluate the infinity integral with an exponential term with second degree

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx$$

(2) **Usage**

Double precision:

ierr = ASL_dh2int (n, f, &w, work);

Single precision:

ierr = ASL_rh2int (n, f, &w, work);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	n	I	1	Input	Number of Gauss points
2	f	–	–	Input	Name defining $f(x)$
3	w	{D∗ / R∗}	1	Output	$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx$
4	work	{D∗ / R∗}	3×n	Work	Work area
5	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
4000	The sequence did not converge in the step where Gauss points were obtained.	
5000	Overflow occurred in evaluating Legendre polynomials.	

(6) **Notes**

(a) f should be defined as **Example**.
double precision

```
void FORTRAN sfun(double *x, double *y)
{
      *y=f(*x);
}
```

single precision

```
void FORTRAN sfun(float *x,  float *y)
{
      *y=f(*x);
}
```

(b) $f(x)$ can be used if it is defined in $-7 < x < 7$. Namely, as

$$\int_{-\infty}^{-7} e^{-x^2} dx + \int_{7}^{\infty} e^{-x^2} dx < 7.415 \cdot 10^{-23},$$

in $x \leq -7$ and $x \geq 7$, the functional value $f(x)$ is omitted to get the relational precision $7.4 \cdot 10^{-16}$.

(7) **Example**

(a) Problem
Evaluate

$$\int_{-\infty}^{\infty} \frac{e^{-x^2}}{x^2+1} dx$$

(true value is $\pi e \operatorname{Erfc}(1)$).

(b) Main program

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void fn(double *x, double *y)
#else
void fn(x, y)
double *x;
double *y;
#endif
{
    *y=1.0/((*x)*(*x)+1.0);
}
#ifdef __cplusplus
}
#endif
#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void fn1(double *x1, double *y1)
#else
void fn1(x1, y1)
double *x1;
double *y1;
#endif
{
    double x,y,z,z2;
    x=*x1;
    z=x;
    if(z<0.0)
    {  z=-z;}
    z2=z*z;
    y=z2*z2;
    y=y*y*z2;
    *y1=y*z;
```

```
}
#ifdef __cplusplus
}
#endif
int main()
{
    int n,ierr;
    double w, *work;
    double v1,v2,one,verfc;
    n=24;
    one=1.0;
    work=(double * )malloc((size_t)( sizeof(double)* (3*n) ));
    if( work == NULL )
    {
        printf( "no enough memory for array work  \n" );
        return -1;
    }
    printf( "\n\t *** ASL_dh2int   \n\n" );
    ierr=ASL_dh2int(n, fn, &w, work);
    printf( "\n\t *** OUTPUT ***\n\n" );
    printf( "\n\tierr = %6d\n", ierr );
    printf( "\n\t *** value 1   true1  \n\n" );
    ierr=ASL_wierfc(1,&one, &verfc);
    v1=M_PI*exp(one)*verfc;
    printf( "\n%13.8g,%13.8g\n",w,v1);
    ierr=ASL_dh2int(n, fn1, &w, work);
    printf( "\n\t *** OUTPUT ***\n\n" );
    printf( "\n\tierr = %6d\n", ierr );
    v2=120;
    printf( "\n\t *** value 2   true2  \n\n" );
    printf( "\n%13.8g,%13.8g\n",w,v2);
    free(work);
    return 0;
}
```

(c) Output results

```
 *** ASL_dh2int


 *** OUTPUT ***


ierr =      0

 *** value 1   true1


1.3432934,    1.3432934

 *** OUTPUT ***


ierr =      0

 *** value 2   true2


      120,          120
```

4.5 INTEGRATION OVER A TWO-DIMENSIONAL FINITE INTERVAL

4.5.1 ASL_dhnrnm, ASL_rhnrnm
Two-Dimensional Integration over a Rectangular Area

(1) **Function**

ASL_dhnrnm or ASL_rhnrnm automatically performs two-dimensional integration over a rectangular area.

(2) **Usage**

Double precision:

```
ierr = ASL_dhnrnm  (f, a, b, c, d, er, ea, idv, &q, &ae, &nev);
```

Single precision:

```
ierr = ASL_rhnrnm  (f, a, b, c, d, er, ea, idv, &q, &ae, &nev);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Name defining the integrand $f(x, y)$
2	a	{D / R}	1	Input	Lower end of the integral interval in the X-axis direction
3	b	{D / R}	1	Input	Upper end of the integral interval in the X-axis direction
4	c	{D / R}	1	Input	Lower end of the integral interval in the Y-axis direction
5	d	{D / R}	1	Input	Upper end of the integral interval in the Y-axis direction
6	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error $\times 64$) (See Section 4.1.1)
7	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
8	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value: 5000)
9	q	{D∗ / R∗}	1	Output	Integral value
10	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
11	nev	I*	1	Output	Number of times integrand is evaluated
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $a < b, c < d$

(b) er $\geq$Unit for determining error $\times 64$
(except when 0.0 is input since the default value is assumed)

(c) ea $\geq$Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) $idv > 1$
(except when 0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value is multiplied by -1 or the integral value = 0. (If d < c and b < a, the integral value is used as it is.)
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv	The minimum subdivision width is gradually widened so that an approximate solution can be obtained.
2400	A certain subdivided interval cannot be further subdivided.	Processing is terminated without obtaining a solution having the required precision.
2500	The solution precision will not reach the require precision.	
3500	The result is unreliable (the error is larger than the result).	Processing is aborted.
4000	Overflow occurred more than once in a single subdivided interval.	

(6) **Notes**

(a) If the integrand has a peak in an extremely narrow range, you should increase the required precision by using the double-precision function when obtaining the solution. An appropriate value for the required relative precision is up to the square root of the value of the unit for determining error.

(b) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(c) This function uses an algorithm based on the adaptive Newton-Cotes 9-point rule but having more powerful singular point processing capabilities and extended to two dimensions.

(7) **Example**

(a) Problem

Obtain the value of $\int_0^2 \int_0^2 (x+y)dxdy$.

(b) Input data

Function name corresponding to integrand $f(x, y)$: f.
a=0.0, b=2.0, c=0.0, d=2.0, er=0.0, ea=0.0 and idv=0.

(c) Main program

```
/*      C interface example for ASL_dhnrnm */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
```

```
double f(double *x,double *y)
#else
double f(x,y)
double *x;
double *y;
#endif
{
    return (*x)+(*y);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    double c;
    double d;
    double epsr;
    double epsa;
    int limi;
    double result;
    double abserr;
    int neval;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnrnm.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnrnm ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &c );
    fscanf( fp, "%lf", &d );
    fscanf( fp, "%lf", &epsr );
    fscanf( fp, "%lf", &epsa );
    fscanf( fp, "%d", &limi );

    printf( "\ta     = %8.3g\n", a );
    printf( "\tb     = %8.3g\n", b );
    printf( "\tc     = %8.3g\n", c );
    printf( "\td     = %8.3g\n", d );
    printf( "\ters   = %8.3g\n", epsr );
    printf( "\teas   = %8.3g\n", epsa );
    printf( "\tidv   =   %6d\n", limi );

    fclose( fp );

    ierr = ASL_dhnrnm(f, a, b, c, d, epsr, epsa, limi, &result, &abserr, &neval);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );
    printf( "\n\tNumber of Function Evaluations\n" );
    printf( "\t  nev = %6d\n", neval );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnrnm ***

 ** Input **

a     =        0
b     =        2
c     =        0
d     =        2
ers   =        0
eas   =        0
idv   =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =        8
```

```
Estimate of Absolute Error
  ae  = 1.78e-14

Number of Function Evaluations
  nev =    441
```

4.5.2 ASL_dhnfnm, ASL_rhnfnm
Two-Dimensional Integration over an Area Indicated by the Function

(1) **Function**

ASL_dhnfnm or ASL_rhnfnm automatically performs two-dimensional integration over an arbitrary area in which the integral range in the X-axis direction is given as a function of y.

Calculate the value of integration $\int_c^d \int_{a(y)}^{b(y)} f(x,y)dxdy$

(2) **Usage**

Double precision:

```
ierr = ASL_dhnfnm  (f, a, b, c, d, er, ea, idv, &q, &ae, &nev);
```

Single precision:

```
ierr = ASL_rhnfnm  (f, a, b, c, d, er, ea, idv, &q, &ae, &nev);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	−	Input	Name defining the integrand $f(x, y)$
2	a	{D / R}	−	Input	Function $a(y)$ giving the lower end of the integral interval in the X-axis direction
3	b	{D / R}	−	Input	Function $b(y)$ giving the upper end of the integral interval in the X-axis direction
4	c	{D / R}	1	Input	Lower end of the integral interval in the Y-axis direction
5	d	{D / R}	1	Input	Upper end of the integral interval in the Y-axis direction
6	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64) (See Section 4.1.1)
7	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}$) (See Section 4.1.1)
8	idv	I	1	Input	Maximum number of subdivided intervals for normal processing (Default value: 5000)
9	q	{D* / R*}	1	Output	Integral value
10	ae	{D* / R*}	1	Output	Estimated value of absolute error (See Section 4.1.1)
11	nev	I*	1	Output	Number of times integrand is evaluated
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) c < d

(b) er ≥Unit for determining error ×64
(except when 0.0 is input since the default value is assumed)

(c) ea ≥Positive minimum value $\times 2^{24}$
(except when 0.0 is input since the default value is assumed)

(d) idv > 1
(except when 0 is input since the default value is assumed)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	Restriction (a) was not satisfied.	The integral value is multiplied by -1 or the integral value = 0.
1500	Restriction (b), (c) or (d) was not satisfied.	Processing is performed with er, ea or idv set to the default value.
2000	Number of times integrand is evaluated reached idv	The minimum subdivision width is gradually widened so that an approximate solution can be obtained.
2400	A certain subdivided interval cannot be further subdivided.	Processing is terminated without obtaining a solution having the required precision
2500	The solution precision will not reach the required precision	
3500	The result is unreliable (the error is larger than the result).	Processing is aborted.
4000	Overflow occurred more than once in a single subdivided interval.	

(6) **Notes**

(a) If the integrand has a peak in an extremely narrow range, you should increase the required precision by using the double-precision function when obtaining the solution. An appropriate value for the required relative precision is up to the square root of the value of the unit for determining error.

(b) If a default value is shown in the "contents" column of the argument table, then the default value is set if 0 is input for an integer-type argument or 0.0 is input for a real-type argument.

(c) This function uses an algorithm based on the adaptive Newton-Cotes 9-point rule but having more powerful singular point processing capabilities and extended to two dimensions.

(7) **Example**

(a) Problem

Obtain the value of $\int_0^2 \int_0^{\frac{\sqrt{4-y^2}}{2}} (x+y)dxdy$.

(b) Input data

Function name corresponding to integrand $f(x)$: f.

Function name corresponding to function giving the lower end and upper end of the integral interval in the X-axis direction: a and b.

c=0.0, d=2.0, er=1.0e−8, ea=0.0 and idv=0.

(c) Main program

```
/*      C interface example for ASL_dhnfnm */

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

#ifdef __cplusplus
extern "C"
{
```

```
#endif
#ifdef __STDC__
double f(double *x,double *y)
#else
double f(x,y)
double *x;
double *y;
#endif
{
    return (*x)+(*y) ;
}
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double a(double *y)
#else
double a(y)
double *y;
#endif
{
    return 0.0*(*y);
}
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double b(double *y)
#else
double b(y)
double *y;
#endif
{
    return 0.5*sqrt(4.0-(*y)*(*y));
}
#ifdef __cplusplus
}
#endif


int main()
{
    double c;
    double d;
    double epsr;
    double epsa;
    int limi;
    double result;
    double abserr;
    int neval;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnfnm.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnfnm ***\n" );
    printf( "\n     ** Input **\n\n" );
    fscanf( fp, "%lf", &c );
    fscanf( fp, "%lf", &d );
    fscanf( fp, "%lf", &epsr );
    fscanf( fp, "%lf", &epsa );
    fscanf( fp, "%d", &limi );

    printf( "\tc     = %8.3g\n", c );
    printf( "\td     = %8.3g\n", d );
    printf( "\ter    = %8.3g\n", epsr );
    printf( "\tea    = %8.3g\n", epsa );
    printf( "\tidv   =   %6d\n", limi );

    fclose( fp );

    ierr = ASL_dhnfnm(f, a, b, c, d, epsr, epsa, limi, &result, &abserr, &neval);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
```

```
        printf( "\n\tIntegral Approximation\n" );
        printf( "\t  q   = %8.3g\n", result );
        printf( "\n\tEstimate of Absolute Error\n" );
        printf( "\t  ae  = %8.3g\n", abserr );
        printf( "\n\tNumber of Function Evaluations\n" );
        printf( "\t  nev = %6d\n", neval );

        return 0;
    }
```

(d) Output results

```
*** ASL_dhnfnm ***

 ** Input **

c     =        0
d     =        2
er    =    1e-08
ea    =        0
idv   =        0

 ** Output **

ierr =      0

Integral Approximation
  q   =        2

Estimate of Absolute Error
  ae  =  1.4e-10

Number of Function Evaluations
  nev =   3990
```

4.6 MULTI-DIMENSIONAL INTEGRATION OVER A FINITE INTERVAL

4.6.1 ASL_dhnrml, ASL_rhnrml
Multi-Dimensional Integration over a Hypercubic Space

(1) **Function**

ASL_dhnrml or ASL_rhnrml performed a multi-dimensional integration of a function over a hypercubic space of more than two dimensions. (If the function to be integrated is two-dimensional and has singularities, a function for the two-dimensional integration is recommended.)

(2) **Usage**

Double precision:

```
ierr = ASL_dhnrml  (f, a, b, m, er, ea, itmx, &q, &ae, &nev, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnrml  (f, a, b, m, er, ea, itmx, &q, &ae, &nev, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Function name of integrand $f(x_1, \cdots, x_m)$
2	a	{D∗ / R∗}	m	Input	a_i is lower limit of integration in x_i-axis direction. $i = 1, 2, \cdots, m$
3	b	{D∗ / R∗}	m	Input	b_i is upper limit of integration in x_i-axis direction. $i = 1, 2, \cdots, m$
4	m	I	1	Input	Dimensionality of integration
5	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64×m) (See Section 4.1.1)
6	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24}\times$ m) (See Section 4.1.1)
7	itmx	I	1	Input	Maximum number of repetitions (Default value: 60/m)
8	q	{D∗ / R∗}	1	Output	Integral value
9	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
10	nev	I*	1	Output	Number of times integrand is evaluated
11	iwk	I*	m	Work	Work area
12	wk	{D∗ / R∗}	3×m	Work	Work area
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $2 \leq$ m ≤ 9

(b) er $\geq$ Unit for determining error $\times 64 \times$ m
(except when 0.0 is input to set the default)

(c) ea $\geq$ Positive minimum value $\times 2^{24} \times$ m
(except when 0.0 is input to set the default)

(d) $4 \leq$ itmx ≤ 30 (except when 0 is input to set the default)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	$a_i > b_i$, $a_i = b_i$	$\int_{a_i}^{b_i} f dx_i = -\int_{b_i}^{a_i} f dx_i$. Changes the integral as above and integrates it. Or the integral is zero.
1500	Restriction (b), (c) or (d) was not satisfied.	Processes with default values.
2500	Repetition was terminated before the required precision of solution was reached.	Terminates processing without obtaining the required precision. (Solution with the best precision at the time is returned.)
3000	Restriction (a) was not satisfied.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	Returns the obtained result.

(6) **Notes**

(a) The function f should be created as follows:

```
double FORTRAN f(double *x, int *m)
{
    return(f(x[0], ···, x[*m − 1]))
}
```

(b) The required relative precision should be approximately $\sqrt[m]{10^{-6}}$ if there is any singularity (a point where a function is not differentiable since its derivative is discontinuous or ∞) in the interval of integration, approximately max($\sqrt[m]{10^{-12}}, 10^{-4}\times$m) if there is any bad singularity (a point where a function becomes ∞) at the limits of integration or approximately $\sqrt{\text{Unit for determining error}} \times \text{m}^2/20$ in other cases.

(c) If a default value is available for an argument, input 0 for the integer type or 0.0 for the real type to set the default value.

(d) This function obtains the value of an integral by accelerating, with the improved algorithm, the solution series which is obtained by increasing n of the N-point Gauss-Romberg rule in each dimension.

(7) **Example**

(a) Problem

Obtain the value of $\int_0^1 \int_0^1 \int_0^1 1/(3-\cos(\pi x)-\cos(\pi y)-\cos(\pi z))dxdydz$.

(b) Input data

Function name corresponding to integrand $f(x_1, \cdots, x_m)$: f.
a[0]=a[1]=a[2]=0.0, b[0]=b[1]=b[2]=1.0, m=3, er=1.0e−4, ea=0.0 and itmx=15.

(c) Main program

```
/*      C interface example for ASL_dhnrml */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x,int *m)
#else
double f(x,m)
double *x;
int   *m;
#endif
{
    return 1.0/(3.0-cos(M_PI*x[0])-cos(M_PI*x[1])-cos(M_PI*x[2]));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double *a;
    double *b;
    int m;
    double epsr;
    double epsa;
    int lim;
    double result;
    double abserr;
    int neval;
    int *iwk;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dhnrml.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnrml ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &m );

    a = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( a == NULL )
    {
        printf( "no enough memory for array a\n" );
        return -1;
    }

    b = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( b == NULL )
    {
        printf( "no enough memory for array b\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (3*m) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    iwk = ( int * )malloc((size_t)( sizeof(int) * m ));
```

```
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }
    for( i=0 ; i<m ; i++ )
    {
        fscanf( fp, "%lf", &a[i] );
        printf( "\ta[%6d]=%8.3g\n", i,a[i] );
    }
    for( i=0 ; i<m ; i++ )
    {
        fscanf( fp, "%lf", &b[i] );
        printf( "\tb[%6d]=%8.3g\n", i,b[i] );
    }
    fscanf( fp, "%lf", &epsr );
    fscanf( fp, "%lf", &epsa );
    fscanf( fp, "%d", &lim );

    printf( "\tm        =  %6d\n", m );
    printf( "\ter       =%8.3g\n", epsr );
    printf( "\tea       =%8.3g\n", epsa );
    printf( "\titmx     =  %6d\n", lim );

    fclose( fp );

    ierr = ASL_dhnrml(f, a, b, m, epsr, epsa, lim, &result, &abserr, &neval, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );

    free( a );
    free( b );
    free( wk );
    free( iwk );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnrml ***

 ** Input **

a[     0]=       0
a[     1]=       0
a[     2]=       0
b[     0]=       1
b[     1]=       1
b[     2]=       1
m        =       3
er       =  0.0001
ea       =       0
itmx     =      15

 ** Output **

ierr =      0

Integral Approximation
  q   =    0.505

Estimate of Absolute Error
  ae  = 1.24e-05
```

4.6.2 ASL_dhnfml, ASL_rhnfml
Multi-Dimensional Integration over a Space Indicated by a Function

(1) **Function**

ASL_dhnfml, ASL_rhnfml performs a multi-dimensional integration of a function over a space indicated by a function of more than two dimensions. (If the function to be integrated is two-dimensional and has singularities, a function for the two-dimensional integration is recommended.)

Performs the following integration

$$\int_{a_m}^{b_m} \int_{a_{m-1}}^{b_{m-1}} \cdots \int_{a_1}^{b_1} f(x_1, \cdots, x_m) dx_1 \cdots dx_m$$

here,

$$a_i = f_i(x_{i+1}, \cdots, x_m),\ b_i = g_i(x_{i+1}, \cdots, x_m); i = 1, 2, \cdots, m-1$$
$$a_m = f_m,\ b_m = g_m$$

(2) **Usage**

Double precision:

```
ierr = ASL_dhnfml  (f, r, m, er, ea, itmx, &q, &ae, &nev, iwk, wk);
```

Single precision:

```
ierr = ASL_rhnfml  (f, r, m, er, ea, itmx, &q, &ae, &nev, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Function name of integrand $f(x_1, \cdots, x_m)$
2	r	–	–	Input	Name of function which gives lower limit a_i and upper limit $b_i (i = 1, \cdots, m)$ of integration
3	m	I	1	Input	Dimensionality of integration
4	er	{D / R}	1	Input	Required relative precision (Default value: Unit for determining error ×64×m) (See Section 4.1.1)
5	ea	{D / R}	1	Input	Required absolute precision (Default value: Positive minimum value $\times 2^{24} \times$ m) (See Section 4.1.1)
6	itmx	I	1	Input	Maximum number of repetitions (Default value: 60/m)
7	q	{D∗ / R∗}	1	Output	Integral value
8	ae	{D∗ / R∗}	1	Output	Estimated value of absolute error (See Section 4.1.1)
9	nev	I*	1	Output	Number of times integrand is evaluated
10	iwk	I*	m	Work	Work area
11	wk	{D∗ / R∗}	3×m	Work	Work area
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $2 \leq$ m ≤ 9

(b) er $\geq$ Unit for determining error $\times 64 \times$ m
(except when 0.0 is input to set the default)

(c) ea $\geq$ Positive minimum value $\times 2^{24} \times$ m
(except when 0.0 is input to set the default)

(d) $4 \leq$ itmx ≤ 30 (except when 0 is input to set the default)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1200	$a_i > b_i$, $a_i = b_i$	$\int_{a_i}^{b_i} f dx_i = -\int_{b_i}^{a_i} f dx_i$. Changes the integral as above and integrates it. Or the integral is zero.
1500	Restriction (b), (c) or (d) was not satisfied.	Processes with default values.
2500	Repetition was terminated before the required precision of solution was reached.	Terminates processing without obtaining the required precision. (Solution with the best precision at the time is returned.)
3000	Restriction (a) was not satisfied.	Processing is aborted.
3500	The result is unreliable (the error is larger than the result).	Returns the obtained result.

(6) **Notes**

(a) The function f should be created as follows:

```
double FORTRAN f(double *x, int *m)
{
    return(f(x[0], ···, x[*m-1]))
}
```

(b) **Function**r is made as follows:

```
void FORTRAN r(int *i, double *x, double *a, double *b, int *m)
{
    if(*i==1)
    {
        a[0]=f_1(x[1], ···, x[m − 1]);
        b[0]=g_1(x[1], ···, x[m − 1]);
    }
    else if(*i==2)
    {
        a[1]=f_2(x[2], ···, x[m − 1]);
        b[1]=g_2(x[2], ···, x[m − 1]);
    }
        ⋮
        ⋮
    else if(*i==(*m−1))
    {
        a[*m−2] = f_{*m−2}(x[m − 1]);
        b[*m−2] = g_{*m−2}(x[m − 1]);
    }
```

else if(∗i==(∗m))
{
$a[* \text{ m}-1] = f_{\text{m}}$;
$b[* \text{ m}-1] = g_{\text{m}}$;
}
}

(c) The required relative precision should be approximately $\sqrt[\text{m}]{10^{-6}}$ if there is any singularity (a point where a function is not differentiable since its derivative is discontinuous or ∞) in the interval of integration, approximately $\max(\sqrt[\text{m}]{10^{-12}}, 10^{-4}\times\text{m})$ if there is any bad singularity (a point where a function becomes ∞) at the limits of integration or approximately $\sqrt{\text{Unit for determining error}} \times \text{m}^2/20$ in other cases.

(d) If a default value is available for an argument, input 0 for the integer type or 0.0 for the real type to set the default value.

(e) This function obtains the value of an integral by accelerating, with the improved θ-algorithm, the solution series which is obtained by increasing n of the N-point Gauss-Romberg rule in each dimension.

(7) **Example**

(a) Problem

$$\int_0^1 \int_0^{\sqrt{1-z^2}} \int_0^{\sqrt{1-y^2-z^2}} \sqrt{1-x^2-y^2-z^2}dxdydz$$

(b) Input data
Function name corresponding to integrand $f(x_1, \cdots, x_m)$: f.
Name of function which gives lower limit and upper limit of integration: r.
m=3, er=1.0e−8, ea=0.0 and itmx=15.

(c) Main program

```
/*      C interface example for ASL_dhnfml */

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

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x,int *m)
#else
double f(x,m)
double *x;
int  *m;
#endif
{
    return sqrt(1.0-x[0]*x[0]-x[1]*x[1]-x[2]*x[2]);
}
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
void    r(int *i,double *x,double *a,double *b,int *m)
#else
void    r(i,x,a,b,m)
double *x,*a,*b;
int   *i,*m;
#endif
{
    if( *i == 1 )
    {
            a[0]=0.0;
            b[0]=sqrt(1.0-x[1]*x[1]-x[2]*x[2]);
```

```
    }
    else if( *i == 2 )
    {
            a[1]=0.0;
            b[1]=sqrt(1.0-x[2]*x[2]);
    }
    else
    {
            a[2]=0.0;
            b[2]=1.0;
    }
}
#ifdef __cplusplus
}
#endif

int main()
{
    int m;
    double epsr;
    double epsa;
    int lim;
    double result;
    double abserr;
    int neval;
    int *iwk;
    double *wk;
    int ierr;
    FILE *fp;

    fp = fopen( "dhnfml.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dhnfml ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &m );
    fscanf( fp, "%lf", &epsr );
    fscanf( fp, "%lf", &epsa );
    fscanf( fp, "%d", &lim );

    printf( "\tm   =   %6d\n", m );
    printf( "\ter  = %8.3g\n", epsr );
    printf( "\tea  = %8.3g\n", epsa );
    printf( "\titmx=   %6d\n", lim );

    fclose( fp );

    wk = ( double * )malloc((size_t)( sizeof(double) * (3*m) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    iwk = ( int * )malloc((size_t)( sizeof(int) * m ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    ierr = ASL_dhnfml(f, r, m, epsr, epsa, lim, &result, &abserr, &neval, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tIntegral Approximation\n" );
    printf( "\t  q   = %8.3g\n", result );
    printf( "\n\tEstimate of Absolute Error\n" );
    printf( "\t  ae  = %8.3g\n", abserr );

    free( wk );
    free( iwk );

    return 0;
}
```

(d) Output results

```
*** ASL_dhnfml ***

 ** Input **

m   =        3
er  =    1e-08
ea  =        0
itmx=       15
```

```
 ** Output **

ierr =      0

Integral Approximation
  q   =    0.308

Estimate of Absolute Error
  ae  = 1.64e-09
```

Chapter 5

APPROXIMATION AND INTERPOLATION

5.1 INTRODUCTION

This chapter describes functions that fit a function to given data points. This library provides the following functions for obtaining the optimum coefficients of the function that approximates user-assigned data points according to a least squares approximation.

(1) Least squares approximation orthogonal polynomials

(2) Least squares approximation nonlinear functions

(3) Two-dimensional arbitrary data least squares approximation polynomials

(4) Two-dimensional lattice data least squares approximation polynomials

The function that obtains the least squares approximation orthogonal polynomials determines the coefficients of an orthogonal polynomial that minimizes the sum of the squares of the differences of the function values y_i and the orthogonal polynomial values for x at $x = x_i$ when function values y_i $(i = 1, 2, \cdots, n)$ at n data points x_i $(i = 1, 2, \cdots, n)$ are given.
The function that obtains the least squares approximation nonlinear functions determines the user-defined function that minimizes the sum of the squares of the differences of the function values y_i and the values of the user-defined function at $x = x_i$ when function values y_i $(i = 1, 2, \cdots, n)$ at n data points x_i $(i = 1, 2, \cdots, n)$ are given.
The function that obtains the two-dimensional arbitrary data least squares approximation polynomials determines the coefficients of a polynomial for x and y that minimizes the sum of the squares of the differences of the function values z_i and the values of the polynomial for x and y at $(x, y) = (x_i, y_i)$ when n two-dimensional coordinate points (x_i, y_i) $(i = 1, 2, \cdots, n)$ and function values z_i at those points are given.
The function that obtains the two-dimensional lattice data least squares approximation polynomials determines the coefficients of a polynomial for x and y that minimizes the sum of the squares of the differences of the function values z_{ij} and the values of the polynomial for x and y at $(x, y) = (x_i, y_i)$ when all function values z_{ij} at $nx \times ny$ two-dimensional lattice points (x_i, y_i) $(i = 1, 2, \cdots, nx;\ j = 1, 2, \cdots, ny)$ are given.
For interpolation, this library provides the following functions for obtaining interpolation values and interpolation polynomial coefficients for user-assigned data points.

(1) Unequally spaced discrete point interpolation value

(2) Unequally spaced discrete point interpolation value and interpolation coefficients

(3) Discrete point interpolation value on two-dimensional cross section lines

(4) Discrete point interpolation value on two-dimensional lattice

The function that obtains the unequally spaced discrete point interpolation value obtains the Y coordinate value at the interpolation point by using Aitken's scheme to interpolate n given data points (x_i, y_i) $(i = 1, 2, \cdots, n)$ for a given interpolation point x.
The function that obtains the unequally spaced discrete point interpolation value and interpolation coefficients

obtains the Y coordinate value at the interpolation point and the interpolation polynomial coefficients by using Newton's method to interpolate n given data points (x_i, y_i) $(i = 1, 2, \cdots, n)$ for a given interpolation point x.
The function that obtains the discrete point interpolation value on two-dimensional cross section lines establishes nx straight lines (called cross section lines here) $x = x_i$ $(i = 1, 2, \cdots, nx)$ parallel to the Y-axis in the XY-plane and obtains the interpolation value at an arbitrary point according to a cubic spline function by assigning ny_i data points on each of those cross sections and function values at those points. This function also obtains spline coefficients used for interpolating data on the given cross section lines.
The function that obtains the discrete point interpolation value on a two-dimensional lattice obtains the interpolation value of the function at a single point (x, y) when all function values z_{ij} on two-dimensional lattice points (x_i, y_j) $(i = 1, 2, \cdots, nx;\ j = 1, 2, \cdots, ny)$ are given.
For a Chebyshev approximation, the following function is provided for obtaining the Chebyshev coefficients of the function that obtains the best fit approximation of a function given by the user.

(1) Chebyshev approximation

The Chebyshev approximation function determines the coefficients of the Chebyshev polynomial that obtains the relative minimum of the difference between the Chebyshev polynomial value and function value y_i for $x = x_i$ $(i = 0, 2, \cdots, n)$ when a function $f(x)$ is given in the finite interval $[a, b]$. That is, if T_k is the Chebyshev polynomial for the Chebyshev coefficients c_k $(k = 0, 1, \cdots, m)$ that were obtained, the polynomial coefficients d_k $(k = 0, 1, \cdots, m)$ are obtained so that

$$\sum_{k=0}^{m}(d_k y^k) = \sum_{k=0}^{m}(c_k T_k(y))$$

Also, the degree of the optimal approximation polynomial is obtained by the this function.

5.1.1 Notes

(1) The coefficient initial values should be taken as close as possible to the optimal coefficients.

(2) An appropriate value for the required precision is on the order of the square root of the unit for determining error.

(3) The appropriate required maximum error for the Chebyshev approximation is on the order of (Chebyshev coefficients truncation error) $\times 10^2$. (See Section 5.6.1)

5.1.2 Algorithms Used

5.1.2.1 Least squares approximation orthogonal polynomials

When the data points (x_i, y_i) and weight function $w(x_i)$ $(i = 1, 2, \cdots, n)$ are given, y_i will be approximated by the following polynomial of degree m :

$$f(x) = a_0 x^m + a_1 x^{m-1} + \cdots + a_{m-1} x + a_m \tag{5.1}$$

If (x_i, y_i) are not distributed in the neighborhood of the origin, then to minimize error, a coordinate conversion is performed for the calculations so that the midpoint of the distribution of x_i will be 0.0 and the minimum value of y_i will be 0.0.
Now, orthogonal polynomials $P_j(x)$ $(j = 0, 1, \cdots, m)$ are defined as polynomials that satisfy the following relationship:

$$\sum_{i=1}^{n} w(x_i) P_u(x_i) P_v(x_i) = \sum_{i=1}^{n} w(x_i) \{P_u(x_i)\}^2 \delta_{uv} \tag{5.2}$$

where δ_{uv} is the Kronecker delta defined as follows:

$$\delta_{uv} = \begin{cases} 1 & (u = v) \\ 0 & (u \neq v) \end{cases}$$

$f(x)$ is expressed as the following linear combination of the orthogonal polynomials $P_j(x)$ $(j = 0, 1, \cdots, m)$:

$$f(x) = \sum_{j=0}^{m} b_j P_j(x) \tag{5.3}$$

(1) Determination of b_j and $P_j(x_i)$ $(i = 1, 2, \cdots, n;\ j = 0, 1, \cdots, m)$
The coefficients b_j are determined according to the least squares method. That is b_j are determined so that the following function:

$$H(b_0, \cdots, b_m) \equiv \sum_{i=1}^{n} w(x_i) \{y_i - f(x_i)\}^2$$

is minimized. This function is minimized when:

$$\frac{\partial H}{\partial b_k} = 0 \tag{5.4}$$

By using the orthogonality condition of (5.2) in (5.4), the coefficients b_j satisfy the following:

$$b_j = \frac{\sum_{i=1}^{n} w(x_i) y_i P_j(x_i)}{\sum_{i=1}^{n} w(x_i) \{P_j(x_i)\}^2} \ (j = 0, 1, \cdots, m) \tag{5.5}$$

The orthogonal polynomials $P_j(x)$ can be constructed according to the following recurrence relations as j-degree polynomials of x :

$$\begin{aligned} P_{-1}(x) &= 0 \\ P_0(x) &= 1 \\ P_{j+1}(x) &= (x - \alpha_{j+1}) P_j(x) - \beta_j P_{j-1}(x) \ (j = 0, \cdots, m-1) \end{aligned} \tag{5.6}$$

where the coefficients α_{j+1} and β_j are expressed as follows:

$$\begin{aligned} \alpha_{j+1} &= \frac{\sum_{i=1}^{n} w(x_i) x_i \{P_j(x_i)\}^2}{\sum_{i=1}^{n} w(x_i) \{P_j(x_i)\}^2} \\ \beta_j &= \frac{\sum_{i=1}^{n} w(x_i) \{P_j(x_i)\}^2}{\sum_{i=1}^{n} w(x_i) \{P_{j-1}(x_i)\}^2} \end{aligned} \tag{5.7}$$

from the orthogonality condition given in (5.2).
The following values can be obtained sequentially from (5.6) and (5.7):

$$P_{-1}(x_i), P_0(x_i) \to \alpha_1, \beta_0 \to P_1(x_i) \to \alpha_2, \beta_1 \to \cdots$$

The coefficients b_j $(j = 0, 1, \cdots, m)$ are obtained by using these values and (5.5).

(2) Determining the coefficients a_k $(k = 0, 1, \cdots, m)$
If $P_j(x)$ is expressed as follows:

$$P_j(x) = \sum_{k=0}^{j} c_{j,k} x^k = c_{j,j} x^j + c_{j,j-1} x^{j-1} + \cdots + c_{j,1} x + c_{j,0}$$

then, $c_{j+1,k+1}$ can be written as:

$$c_{j+1,k+1} = c_{j,k} - \alpha_{j+1} c_{j,k+1} - \beta_j c_{j-1,k+1} \ (j = 0, 1, \cdots, m-1;\ k = 0, 1, \cdots, j)$$

(Where, $c_{0,0} = 1$, $c_{-1,0} = 0$).
The value of $c_{j,k}$ are obtained from the above. Also, since $f(x)$ can be written as:

$$f(x) = \sum_{j=0}^{m} b_j \sum_{k=0}^{j} c_{j,k} x^k = \sum_{k=0}^{m} \left(\sum_{j=k}^{m} b_j c_{j,k} \right) x^k$$

the coefficients a_k can be obtained as follows:

$$a_{m-k+1} = \sum_{j=k}^{m} b_j c_{j,k}$$

from the b_j and $c_{j,k}$ that had been obtained as described earlier.

5.1.2.2 Nonlinear least square method

Given n coordinate values (x_i, y_i) $(i = 1, \cdots, n)$ and an approximation function $f(x, \boldsymbol{a})$ that has m parameters $\boldsymbol{a} = \{a_i\}$ $(i = 1, \cdots, m)$, this algorithm obtains a values $\boldsymbol{a}$ for which the following sum of the squares:

$$S(\boldsymbol{a}) = \sum_{i=1}^{n} (y_i - f(x_i, \boldsymbol{a}))^2$$

is a minimum, where the vector function $\boldsymbol{h}(\boldsymbol{a}) = \{h_i(\boldsymbol{a})\}$ is defined as follows:

$$h_i(\boldsymbol{a}) = y_i - f(x_i, \boldsymbol{a}) \ (i = 1, \cdots, n)$$

$S(\boldsymbol{a})$ can be written as:

$$S(\boldsymbol{a}) = \|\boldsymbol{h}(\boldsymbol{a})\|_2^2$$

where $\|\boldsymbol{a}\| = \sqrt{\boldsymbol{a}^T\boldsymbol{a}}$ (the notation ^{T} indicates transpose).
To obtain a solution that minimizes $S(\boldsymbol{a})$, a search is begun from an initial value $\boldsymbol{a} = \boldsymbol{a}_0$, and Powell's hybrid method is used to sequentially correct the solutions $\boldsymbol{a} = \boldsymbol{a}_1, \boldsymbol{a}_2, \cdots$. The hybrid method determines the correction vector at each step as a linear combination of the correction vectors according the steepest descent method and the Gauss-Newton method obtained by linearizing a nonlinear function for the coefficient $\boldsymbol{a}$. An independence check always is performed for the correction vector so that they do not end up being confined within a subspace. Also, the value of the Jacobian matrix is determined from function information and the value from the previous step, without directly calculating the Jacobian matrix in each step.

(1) Correction vector $\Delta\boldsymbol{a}$ calculation
If the vector function $\boldsymbol{h}(\boldsymbol{a} + \Delta\boldsymbol{a})$ is approximated in a linear range from $\Delta\boldsymbol{a}$ as follows:

$$\boldsymbol{h}_L(\boldsymbol{a} + \Delta\boldsymbol{a}) = \boldsymbol{h}(\boldsymbol{a}) + A\Delta\boldsymbol{a}$$

then consider the problem of minimizing:

$$S_L(\boldsymbol{a} + \Delta\boldsymbol{a}) = \|\boldsymbol{h}(\boldsymbol{a}) + A\Delta\boldsymbol{a}\|_2^2$$

where A is the Jacobian matrix $\dfrac{\partial \boldsymbol{h}}{\partial \boldsymbol{a}}$ of $\boldsymbol{h}$.
The correction vector $\Delta\boldsymbol{a}_S$ at each step according to the steepest descent method is given by:

$$\Delta\boldsymbol{a}_S = \frac{\|\boldsymbol{b}\|_2^2}{\|A\boldsymbol{b}\|_2^2}\boldsymbol{b}$$

where $\boldsymbol{b} = -A^T\boldsymbol{h}(\boldsymbol{a})$.
The correction vector $\Delta\boldsymbol{a}_G$ at each step according to the Gauss-Newton method is obtained by solving the following normal equation:

$$A^TA\Delta\boldsymbol{a}_G = \boldsymbol{b}$$

This library uses the QR decomposition algorithm to solve this equation.
In general, the following relationship holds:

$$\|\Delta\boldsymbol{a}_S\|_2 \leq \|\Delta\boldsymbol{a}_G\|_2$$

The correction vector at each step is determined as follows by taking a linear combination of $\Delta\boldsymbol{a}_S$ and $\Delta\boldsymbol{a}_G$ depending on the step size d.

(a) If $d \leq \|\Delta\boldsymbol{a}_S\|_2$, then:

$$\Delta\boldsymbol{a} = d\frac{\Delta\boldsymbol{a}_S}{\|\Delta\boldsymbol{a}_S\|_2}$$

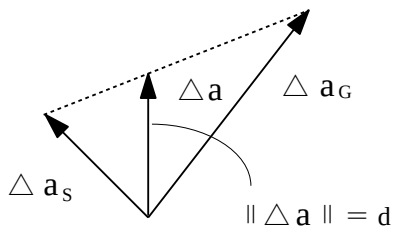


Figure 5−1

(b) If $\|\Delta \boldsymbol{a}_S\|_2 < d < \|\Delta \boldsymbol{a}_G\|_2$, (See Figure 5−1) then:

$$\Delta \boldsymbol{a} = \alpha \Delta \boldsymbol{a}_S + \beta \Delta \boldsymbol{a}_G \ (\alpha > 0, \beta > 0, \|\Delta \boldsymbol{a}\|_2 = d)$$

(c) If $\|\Delta \boldsymbol{a}_G\|_2 \leq d$

$$\Delta \boldsymbol{a} = \Delta \boldsymbol{a}_G$$

(2) Step size d determination

The initial value of the step is assumed to be $d = \|\Delta \boldsymbol{a}_S\|_2$. Thereafter, if the nonlinearity of the function is strong, then the step size is decreased; if the function is nearly linear, then the step size is increased.

To measure the degree of nonlinearity, the ratio $r = \dfrac{\Delta S}{\Delta S_L}$ is used where the amount of change of S in the linear approximation is represented by:

$$\Delta S_L = S_L(\boldsymbol{a} + \Delta \boldsymbol{a}) - S_L(\boldsymbol{a})$$

and the actual amount of change is represented by:

$$\Delta S = S(\boldsymbol{a} + \Delta \boldsymbol{a}) - S(\boldsymbol{a})$$

(a) If $r < 0.1$, then nonlinearity is considered to be strong and d is reduced by half.

(b) If $r \geq 0.1$, then λ, which is the rate of increase of d, is calculated as follows:

$$\lambda^2 = 1.0 - (r - 0.1)\frac{\Delta S_L}{(S_P + \sqrt{(S_P^2 - S_S(r - 0.1)\Delta S_L)}}$$

where, for $\delta \boldsymbol{h}$ defined as:

$$\delta \boldsymbol{h} = \boldsymbol{h}(\boldsymbol{a} + \Delta \boldsymbol{a}) - (\boldsymbol{h}(\boldsymbol{a}) + A\Delta \boldsymbol{a})$$

S_P and S_S are as follows:

$$\begin{aligned} S_P &= \sum_{i=1}^{n} |h_i(\boldsymbol{a} + \Delta \boldsymbol{a})\delta h_i| \\ S_S &= \|\delta \boldsymbol{h}\|_2^2 \end{aligned}$$

Actually, to prevent the value of d from oscillating, d is increased only when an increase is required two times consecutively. Also, the rate of increase is held to at most 2. The actual rate of increase μ is calculated as follows:

$$\begin{aligned} \mu &= \min(2, \lambda, \tau) \\ \tau &= \frac{\lambda}{\mu} \end{aligned}$$

where the initial value for τ is 1, and 1 is reset whenever a reduction of d is required.
In addition, an upper limit d_{max} and lower limit d_{min} are set for d and d is controlled so that it falls between these values.

(3) Correction vector independence check
To correct the Jacobian matrix efficiently, the correction vectors that are taken sequentially must be nearly mutually orthogonal. Therefore, an original independence concept is defined according to a hybrid method, and the correction vectors are controlled so that they are taken in directions that are as independent as possible. According to the hybrid method, a vector $\boldsymbol{p}$ is said to be independent of the i vectors $(\boldsymbol{p}_1, \boldsymbol{p}_2, \cdots, \boldsymbol{p}_i)$ if $\boldsymbol{p}$ forms an angle of at least 30 degrees with an arbitrary vector of the space defined by these i vectors. The calculation for this independence test conceived by Powell is as follows.
m mutually independent vectors from the vectors used to correct the Jacobian matrix during the past $2m$ iterations are made to be orthogonal and are stored in $\Omega = (\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \cdots, \boldsymbol{\omega}_m)$. The array $\boldsymbol{j}$ of size n is used to store information indicating the number of iterations earlier in which $\boldsymbol{\omega}_i$ was the correction vector. That is, this information indicates that $\boldsymbol{\omega}_i$ is the vector that was taken j_i iterations earlier. Ω is initialized as the unit matrix, and $\boldsymbol{j}$ is initialized with values $j_i = m - i + 1$ $(i = 1, \cdots, m)$.
When a solution is corrected, the following steps are performed.

(a) When $\Delta\boldsymbol{a} = \Delta\boldsymbol{a}_G$
Regardless of its independence, $\Delta\boldsymbol{a}$ is taken as the correction vector.

(b) When $\Delta\boldsymbol{a} = \Delta\boldsymbol{a}_G$ does not occur
If $j_1 < 2m$ or if $\Delta\boldsymbol{a}$ is independent of $(\boldsymbol{\omega}_2, \boldsymbol{\omega}_3, \cdots, \boldsymbol{\omega}_m)$, then is taken as the correction vector. Otherwise, the solution is not corrected.

(4) Calculation of Jacobian matrix A
The Jacobian matrix is obtained according to a difference only for the first iteration, and thereafter, it is sequentially updated. This method, which was devised by Broyden, calculates the Jacobian matrix according to the following equation:

$$A' = A + \delta\boldsymbol{h}\frac{\Delta\boldsymbol{a}^T}{\|\Delta\boldsymbol{a}\|_2^2}$$

However, if $\|\Delta\boldsymbol{a}\|_2 < d_{min}$ or if $j_1 = 2m$ and $\Delta\boldsymbol{a}$ is not independent of $(\boldsymbol{\omega}_2, \boldsymbol{\omega}_3, \cdots, \boldsymbol{\omega}_m)$, then $\Delta\boldsymbol{a} = d_{min}\boldsymbol{\omega}_1$ is set.

(5) Revision of Ω and $\boldsymbol{j}$
Ω and $\boldsymbol{j}$ are revised as follows.
When $\Delta\boldsymbol{a} = d_{min}\boldsymbol{\omega}_1$ has been set, the following values should be set:

$$\begin{aligned}
\boldsymbol{\omega}_i &= \boldsymbol{\omega}_{i+1} \qquad (i = 1, \cdots, m-1) \\
\boldsymbol{\omega}_m &= \boldsymbol{\omega}_1 \\
j_i &= j_{i+1} + 1 \qquad (i = 1, \cdots, m-1) \\
j_m &= 1
\end{aligned}$$

Otherwise, the following is performed.
The minimum value k is obtained for which $(\boldsymbol{\omega}_{k+1}, \boldsymbol{\omega}_{k+2}, \cdots, \boldsymbol{\omega}_m, \Delta\boldsymbol{a})$ are mutually independent.
$(\boldsymbol{\omega}_1, \cdots, \boldsymbol{\omega}_{k-1}, \boldsymbol{\omega}_{k+1}, \boldsymbol{\omega}_{k+2}, \cdots, \boldsymbol{\omega}_m, \Delta\boldsymbol{a})$ are made to be orthogonal, and these are again assumed to be

$(\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \cdots, \boldsymbol{\omega}_m)$. The following values are then set:

$$\begin{aligned} j_i &= j_i + 1 \qquad (i = 1, \cdots, k-1) \\ j_i &= j_{i+1} + 1 \quad (i = k, \cdots, m-1) \\ j_m &= 1 \end{aligned}$$

In this way, the relationship $j_1 \leq 2m$ always holds.

(6) Convergence decision
Convergence is assumed to have occurred when the following relationship holds and $\boldsymbol{a} + \Delta\boldsymbol{a}$ is assumed to be the solution:

$$\|\Delta\boldsymbol{a}\|_\infty \leq e_r \max(1, \|\boldsymbol{a} + \Delta\boldsymbol{a}\|_\infty)$$

where e_r is the required relative precision, and:

$$\|\boldsymbol{a}\|_\infty = \max_i |a_i|$$

5.1.2.3 Two-dimensional arbitrary data least squares approximation polynomials

Obtain the following least squares approximation polynomial for given discrete points (x_k, y_k, z_k) $(k = 1, 2, \cdots, n)$ in a space:

$$f(x, y) = \sum_{j=1}^{m+1} \sum_{i=1}^{m+2-j} a_{i,j} x^{i-1} y^{j-1}$$

where, $a_{i,j}$ are the coefficients of the polynomial, x and y are independent variables, and m is the maximum degree of the polynomial for x and y.
Let the residual sum of squares of z_k and $f(x_k, y_k)$ be represented by χ^2, where χ^2 is given by:

$$\chi^2 = \sum_{k=1}^{n} \left(z_k - \sum_{j=1}^{m+1} \sum_{i=1}^{m+2-j} a_{i,j} x_k^{i-1} y_k^{j-1} \right)^2$$

The coefficients $a_{i,j}$ are determined by the following condition for minimizing χ^2 :

$$\frac{\partial \chi^2}{\partial a_{i,j}} = 0$$

Obtain the following system of normal equations from this:

$$\sum_{jc=1}^{m+1} \sum_{ic=1}^{m+2-jc} \sum_{k=1}^{n} a_{ic,jc} x_k^{ic+ir-2} y_k^{jc+jr-2} = \sum_{k=1}^{n} x_k^{ir-1} y_k^{jr-1} z_k$$
$$(jr = 1, \cdots, m+1;\ ir = 1, \cdots, m+2-jr)$$

Now, define G_k, $\boldsymbol{a}$ and $\boldsymbol{b}_k$ as:

$$\begin{aligned} G_k &= (g_{k,(jr,ir),(jc,ic)}) \\ &= (x_k^{ic+ir-2} y_k^{jc+jr-2}) \qquad \begin{pmatrix} jr = 1, \cdots, m+1; & jc = 1, \cdots, m+1 \\ ir = 1, \cdots, m+2-jr; & ic = 1, \cdots, m+2-jc \end{pmatrix} \\ \boldsymbol{a} &= (a_{(jc,ic)}) \qquad \begin{pmatrix} jc = 1, \cdots, m+1 \\ ic = 1, \cdots, m+2-jc \end{pmatrix} \\ \boldsymbol{b}_k &= (b_{k,(jr,ir)}) = (x_k^{ir-1} y_k^{jr-1} z_k) \qquad \begin{pmatrix} jr = 1, \cdots, m+1 \\ ir = 1, \cdots, m+2-jr \end{pmatrix} \end{aligned}$$

and G and $\boldsymbol{b}$ as:

$$G = \sum_{k=1}^{n} G_k$$
$$\boldsymbol{b} = \sum_{k=1}^{n} \boldsymbol{b}_k$$

Then, the system of normal equations is expressed as:

$$G\boldsymbol{a} = \boldsymbol{b}$$

Set $\tilde{G}\tilde{\boldsymbol{a}} = \tilde{\boldsymbol{b}}$ by transforming $\tilde{G}$ to an $\{\frac{(m+1)(m+2)}{2}\}$-order square matrix and $\tilde{\boldsymbol{a}}$ and $\tilde{\boldsymbol{b}}$ to $\{\frac{(m+1)(m+2)}{2}\}$-order column vectors, and obtain the desired polynomial by solving this for $\tilde{\boldsymbol{a}}$.
The polynomial coefficients $a_{i,j}$ are stored in a one-dimensional matrix as shown in Figure 5−2 for computational convenience. The subscript correspondence relationship is expressed as:

$$a(i,j) = \tilde{a}(i + ((j-1) \times (2 \times m - j + 4))/2)$$

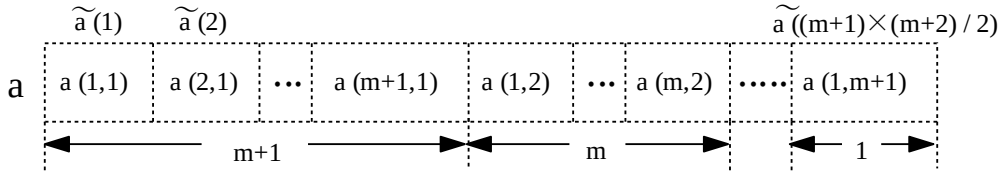


Figure 5−2

5.1.2.4 Two-dimensional lattice data least squares approximation polynomials

If all Z coordinate values $Z_{i,j}$ on two-dimensional lattice points (x_i, y_j) $(i = 1, 2, \cdots, nx;\ j = 1, 2, \cdots, ny)$ are given, obtain the following least squares approximation polynomial that approximates this function:

$$f(x,y) = \sum_{i=1}^{ix+1} \sum_{j=1}^{iy+1} a_{i,j} x^{i-1} y^{j-1} \tag{5.8}$$

Assume that $f(x,y)$ for determining the coefficients $a_{i,j}$ is expressed as follows as a linear combination of the products of the orthogonal polynomials $\Phi_{r-1}(x)$ and $\Psi_{s-1}(y)$, which are defined below:

$$f(x,y) = \sum_{r=1}^{ix+1} \sum_{s=1}^{iy+1} \Gamma_{rs} \Phi_{r-1}(x) \Psi_{s-1}(y) \tag{5.9}$$

Since $\Phi_{r-1}(x)$ and $\Psi_{s-1}(y)$ are $(r-1)$-degree and $(s-1)$-degree polynomials for x and y, respectively, if we let $C_{r,k}$ and $B_{s,l}$ be coefficients of the orthogonal polynomials for x and y, then $\Phi_{r-1}(x)$ and $\Psi_{s-1}(y)$ can be written as the following expressions:

$$\Phi_{r-1}(x) = \sum_{k=1}^{r} C_{r,k} x^{k-1} \tag{5.10}$$

$$\Psi_{s-1}(y) = \sum_{l=1}^{s} B_{s,l} y^{l-1} \tag{5.11}$$

Substitute (5.10) and (5.11) in (5.9) as follows:

$$
\begin{aligned}
f(x,y) &= \sum_{r=1}^{ix+1}\sum_{s=1}^{iy+1}\Gamma_{rs}\Phi_{r-1}(x)\Psi_{s-1}(y) \qquad (5.12)\\
&= \sum_{r=1}^{ix+1}\sum_{s=1}^{iy+1}\Gamma_{rs}\sum_{k=1}^{r}C_{r,k}x^{k-1}\sum_{l=1}^{s}B_{s,l}y^{l-1}\\
&= \sum_{r=1}^{ix+1}\sum_{s=1}^{iy+1}\Gamma_{rs}\sum_{k=r}^{ix+1}C_{r,k}x^{r-1}\sum_{l=s}^{iy+1}B_{s,l}y^{l-1} \quad \text{(See Figure 5−3)}\\
f(x,y) &= \sum_{i=1}^{ix+1}\sum_{j=1}^{iy+1}\left(\Gamma_{ij}\sum_{k=1}^{ix+1}\sum_{l=j}^{iy+1}C_{i,k}B_{j,l}\right)x^{i-1}y^{j-1}
\end{aligned}
$$

By comparing this with (5.8), we obtain the coefficients $a_{i,j}$ as follows:

$$a_{i,j} = \Gamma_{ij}\sum_{k=1}^{ix+1}\sum_{l=j}^{iy+1}C_{i,k}B_{j,l}$$

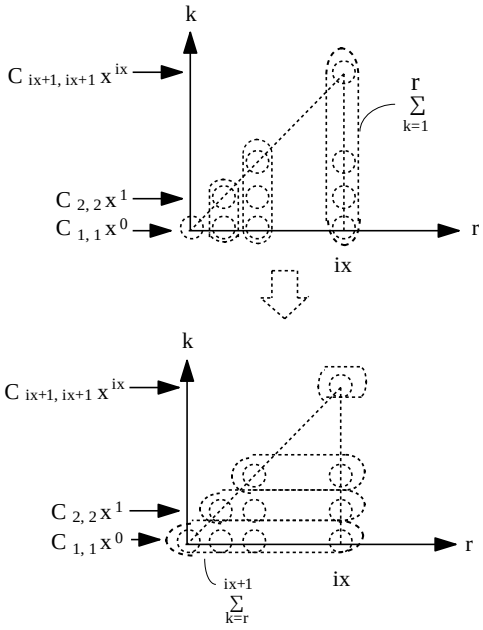


Figure 5−3

(1) Orthogonal polynomials

Orthogonal polynomials for x and y are considered here. Define polynomials that satisfy the following conditions:

$$\sum_{i=1}^{nx}\Phi_r(x_i)\Phi_s(x_i) = D_r\delta_{rs} \qquad (5.13)$$

$$\sum_{j=1}^{ny}\Psi_r(y_j)\Psi_s(y_j) = d_r\delta_{rs} \qquad (5.14)$$

at data points x_i $(i = 1, 2, \cdots, nx)$ and y_j $(j = 1, 2, \cdots, ny)$.
Where δ_{rs} is the Kronecker delta defined as follows:

$$\delta_{rs} = \begin{cases} 1 & (r = s) \\ 0 & (r \neq s) \end{cases}$$

The orthogonal polynomials $\Phi_r(x)$ and $\Psi_s(y)$ for x and y that satisfy the orthogonality conditions shown above can be obtained from the following recurrence relations.
Orthogonal polynomial of x

$$\Phi_r(x) = (x - \alpha_{r-1})\Phi_{r-1}(x) - \beta_{r-1}\Phi_{r-2}(x) \ (r = 2, 3, \cdots) \tag{5.15}$$

where,

$$\begin{aligned} \alpha_{r-1} &= \sum_{i=1}^{nx} x_i \frac{\Phi_{r-1}^2(x_i)}{D_{r-1}} \\ \beta_{r-1} &= \frac{D_{r-1}}{D_{r-2}} \\ \Phi_0(x) &= 1 \\ \Phi_1(x) &= x - \bar{x} \\ \bar{x} &= \sum_{i=1}^{nx} \frac{x_i}{nx} \end{aligned}$$

Orthogonal polynomial of y

$$\Psi_s(y) = (y - \alpha'_{s-1})\Psi_{s-1}(y) - \beta'_{s-1}\Psi_{s-2}(y) \ (s = 2, 3, \cdots) \tag{5.16}$$

where,

$$\begin{aligned} \alpha'_{s-1} &= \sum_{j=1}^{ny} y_j \frac{\Psi_{s-1}^2(y_j)}{d_{s-1}} \\ \beta'_{s-1} &= \frac{d_{s-1}}{d_{s-2}} \\ \Psi_0(y) &= 1 \\ \Psi_1(y) &= y - \bar{y} \\ \bar{y} &= \sum_{j=1}^{ny} \frac{y_j}{ny} \end{aligned}$$

(2) Calculation of Γ_{rs}
First, fix $y = y_j$ and obtain the least squares curved line for x.
Write the least squares curved line as follows using coefficients $\lambda_{r,j}$:

$$q_j(x) \equiv f(x, y_j) = \sum_{r=1}^{ix+1} \lambda_{r,j}\Phi_{r-1}(x) \ (j = 1, 2, \cdots, ny) \tag{5.17}$$

Minimize the residual sum of squares at lattice point $x = x_i$, which is given by the following expression:

$$Q = \sum_{i=1}^{nx} \{q_j(x_i) - z_{i,j}\}^2 \tag{5.18}$$

That is, let $\dfrac{\partial Q}{\partial \lambda_{r,j}} = 0$ and obtain the following expression by using the orthogonality conditions given in (5.13):

$$\lambda_{r,j} = \sum_{i=1}^{nx} z_{i,j} \frac{\Phi_{r-1}(x_i)}{D_{r-1}} \ (r = 1, 2, \cdots, ix + 1) \tag{5.19}$$

By comparing (5.9) with (5.17), we obtain the coefficients as follows:

$$\lambda_{r,j} = \sum_{s=1}^{iy+1} \Gamma_{r,s} \Psi_{s-1}(y_j) \tag{5.20}$$

Multiply both sides of (5.20) by $\Psi_{s-1}(y_j)$ and take the sum from $j = 1$ to ny. Use the orthogonality conditions given in (5.14) to obtain:

$$\Gamma_{rs} = \sum_{j=1}^{ny} \lambda_{r,j} \frac{\Psi_{s-1}(y_j)}{d_{s-1}} \ (s = 1, 2, \cdots, iy + 1) \tag{5.21}$$

Γ_{rs} is determined by using (5.21).

(3) Calculation of $C_{r,k}$ and $B_{s,l}$
Obtain the following recurrence relation for $C_{r,k}$ by substituting (5.10) in (5.15) and setting the coefficients of the powers of x to zero so that this holds for an arbitrary x :

$$C_{r,k} = C_{r-1,k-1} - \alpha_{r-1} C_{r-1,k} - \beta_{r-1} C_{r-2,k}$$

(where $C_{r,k} = 0$ when $r < k$ and $C_{0,0} = 1$).
Similarly, the recurrence relation for $B_{s,l}$ is given as follows:

$$B_{s,l} = B_{s-1,l-1} - \alpha'_{s-1} B_{s-1,l} - \beta'_{s-1} B_{s-2,l}$$

(where $B_{s,l} = 0$ when $s < l$ and $B_{0,0} = 1$).
These expressions can be used to calculate $C_{r,k}$ and $B_{s,l}$.

5.1.2.5 Unequally spaced discrete point interpolation value

If data points (x_i, y_i) $(i = 1, \cdots, n)$ and interpolation point $x = u$ are given, obtain (u_i, v_i) by rearranging the x_i in ascending order of distance from u and simultaneously rearranging the y_i corresponding to them.
To obtain an interpolation polynomial by interpolating two points (u_1, v_1) and (u_2, v_2), use the following basic linear interpolation formula:

$$y(x) = \frac{v_1(x - u_2) - v_2(x - u_1)}{u_1 - u_2} \tag{5.22}$$

to obtain the interpolation polynomial of the first degree.
Also, to obtain an interpolation polynomial by interpolating three points $(u_1, v_1), (u_2, v_2)$, and (u_3, v_3), first perform a linear interpolation between (u_1, v_1) and (u_2, v_2) by using (5.22) to obtain the following first degree interpolation polynomial:

$$y_1^1(x) = \frac{v_1(x - u_2) - v_2(x - u_1)}{u_1 - u_2}$$

Next, perform a linear interpolation between (u_2, v_2) and (u_3, v_3) by using (5.22) to obtain the following first degree interpolation polynomial:

$$y_1^2(x) = \frac{v_2(x-u_3) - v_3(x-u_2)}{u_2 - u_3}$$

By using (5.22) to perform a linear interpolation between the $y_1^1(x)$ and $y_1^2(x)$ that were obtained, the following second degree interpolation polynomial is obtained:

$$y_2^1(x) = \frac{y_1^1(x)(x-u_3) - y_1^2(x)(x-u_1)}{u_1 - u_3}$$

The third degree interpolation polynomial, fourth degree interpolation polynomial, and so on can be created by repeatedly using (5.22) in a similar manner.
In general, the j-th degree interpolation polynomial (where, $j = 2, \cdots, n$) is expressed as follows:

$$y_1^k(x) = \frac{v_k(x-u_{k+1}) - v_{k+1}(x-u_k)}{u_k - u_{k+1}} \ (k = 1, \cdots, j) \tag{5.23}$$

$$y_m^k(x) = \frac{y_{m-1}^k(x)(x-u_{k+m}) - y_{m-1}^{k+1}(x)(x-u_k)}{u_k - u_{k+m}} \tag{5.24}$$
$$(m = 2, \cdots, j;\ k = 1, \cdots, j+1-m)$$

Therefore, the value of the n-th degree interpolation polynomial for $x = u$ (interpolation value) is obtained by calculating (5.23) and (5.24) for $x = u$.
Next, we will explain how to determine the degree of the interpolation polynomial.
First, let Z_j $(= y_j^1(u))$ be the interpolation value that was interpolated using j points in the vicinity of $x = u$.
Define the interpolation difference D_j as follows:

$$D_j \equiv |Z_{j-1} - Z_j| \ (j = 2, \cdots, n)$$

and calculate $D_2, \cdots, D_n$. If the following relationship always holds for the user-assigned required absolute precision ε,

$$D_j > \varepsilon \ (j = 2, \cdots, n)$$

let the value l for which the following holds be the degree of the interpolation polynomial:

$$D_l = \min_j(D_j)$$

and let Z_l be the interpolation value at that time. Also, D_l is output as the absolute error of the interpolation value.
If the following relationship holds for some j $(j \leq n)$,

$$D_j \leq \varepsilon$$

let the minimum j for which this relationship holds be the degree of the interpolation polynomial, and let Z_j be the interpolation value at that time. Also, D_j is output as the absolute error of the interpolation value.

5.1.2.6 Unequally spaced discrete point interpolation value and interpolation coefficients

If data points (x_i, y_i) $(i = 1, \cdots, n)$ and interpolation point $x = u$ are given, obtain (u_i, v_i) by rearranging the x_i in ascending order of distance from u and simultaneously rearranging the y_i corresponding to them.
Let the m-th degree polynomial be expressed as follows:

$$f(x) = c_1 + c_2(x-u_1) + \cdots + c_{m+1}(x-u_1)\cdots(x-u_m) \ (1 \leq m \leq n-1) \tag{5.25}$$

and obtain the coefficients $c_1, \cdots, c_{m+1}$.
Now, if interpolation points $x = u_1, \cdots, u_n$ and function $f(x)$ are given, define divided difference of $f(x)$ inductively as follows:

$$\begin{aligned} f[i] &= f(u_i)\ (i = 1, \cdots, n) \\ f[i_1, \cdots, i_{k+1}] &= \frac{f[i_2, \cdots, i_{k+1}] - f[i_1, \cdots, i_k]}{u_{i_{k+1}} - u_{i_1}}\ (k = 1, \cdots, n) \end{aligned}$$

($i_1, \cdots, i_{k+1}$ are positive integers, they are different from each other, and they are equal to or smaller than n.)
If we set $x = u_1$ in (5.25), we get:

$$c_1 = f(u_1) = f[1]$$

If we substitute this in (5.25) and rearrange the terms, we get:

$$\frac{f(x) - f[1]}{x - u_1} = c_2 + c_3(x - u_2) + \cdots + c_{m+1}(x - u_2) \cdots (x - u_m)$$

If we set $x = u_2$, we get:

$$c_2 = \frac{f(u_2) - f[1]}{u_2 - u_1} = \frac{f[2] - f[1]}{u_2 - u_1} = f[1, 2]$$

If we repeat the above operations in a similar manner, we can represent the coefficients c_i as:

$$c_i = f[1, \cdots, i]\ (i = 1, \cdots, m + 1)$$

Therefore, we can obtain c_i $(i = 1, 2, \cdots, m + 1)$ by sequentially calculating the divided difference and we can calculate the interpolation value at the given interpolation point from these and expression (5.25).

5.1.2.7 Discrete point interpolation value on two-dimensional cross section lines

Consider nx straight lines $x = x_i$ $(i = 1, 2, \cdots, nx)$ perpendicular to the X-axis in the XY-plane. If ny_i $(i = 1, \cdots, nx)$ points $(x_i, y_{j,i})$ $(i = 1, 2, \cdots, nx;\ j = 1, 2, \cdots, ny_i)$ on each straight line and Z coordinate values $z_{i,j}$ $(i = 1, \cdots, nx;\ j = 1, \cdots, ny_i)$ at each point are given, obtain the interpolation value at the point (x_l, y_l) $(x_1 \leq x_l \leq x_{nx};\ \min(y_{j,i}) \leq y_l \leq \max(y_{j,i}))$ as follows. (See Figure 5−4)

(Step 1)

On each cross section, obtain the cubic spline by treating the Y coordinate value of the data point as the abscissa and the function value as the ordinate. (See Section 6.1.2) Using the spline coefficients that were just obtained, obtain interpolation values at the intersections of each cross section line and the straight line $y = y_l$. (See Section 6.1.2) The value may be an extrapolation value depending on the value of y_l.

(Step 2)

Next, in exactly the same way, obtain the spline coefficients by treating the X coordinate value on the cross section line as the abscissa and the interpolation value on the cross section line at the y_l that was just obtained as the ordinate, and obtain the interpolation value at $x = x_l$. The interpolation value at the point (x_l, y_l) can be determined as described above.

5.1.2.8 Discrete point interpolation value on two-dimensional lattice

If all Z coordinate values $z_{i,j}$ on two-dimensional lattice points (x_i, y_j) $(i = 1, \cdots, nx;\ j = 1, \cdots, ny)$ are given, use Aitken's scheme to obtain the interpolation value for an arbitrary point (x_l, y_l) within the lattice. (See Section

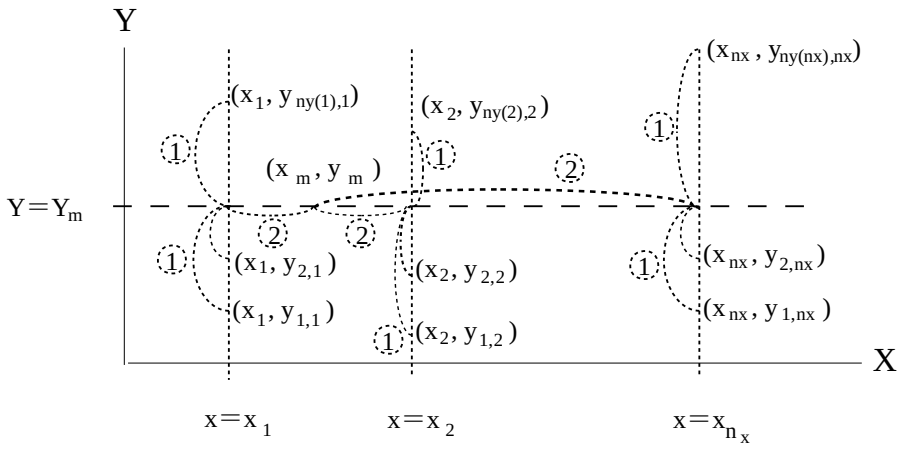


Figure 5−4

5.1.2)
First, obtain the interpolation value at x_l by treating the X coordinate value on each line parallel to the X-axis as the abscissa and the function value as the ordinate. Next, obtain the interpolation value at y_l by treating the Y coordinate value as the abscissa and the interpolation value at x_l that was just obtained as the ordinate.

5.1.2.9 Chebyshev approximation

The norm of the continuous function of $f(x)$ on the closed interval $[a, b]$ is defined by

$$\|f\| = \max_{x\in[a,b]} |f(x)|$$

This is called the maximum value norm or uniform norm.
If we look at polynomials $P_n(x) = a_0 + a_1x + \cdots + a_nx^n$ of degree n (or a rational expression in which the numerator and denominator are polynomials of degree m and n, respectively) as approximations of the function $f(x)$, the polynomial (rational expression) that minimizes the degree of approximation $\|f - P_n\|$ is called the Chebyshev approximation or simply the best (fit) approximation.
Although the best (fit) approximation is good for representing an approximation as a formula, when the procedure for obtaining the approximation is complicated or new calculations are required to add terms, skill and a great deal of effort are required to create the approximation. Although various techniques can be used in the best (fit) approximation, the technique that uses the Chebyshev expansion described below can be calculated relatively easily.
The polynomial conditions for the best (fit) approximation are that the relative maximum values of the error functions are equal and their signs are represented by alternating plus and minus signs. The Chebyshev polynomial satisfies these conditions. The method of sequentially obtaining the Chebyshev polynomial is described below.

(1) Obtaining the Chebyshev coefficients
The Chebyshev polynomial of degree n, which is written as $T_n(x)$, is given by the following explicit function.

$$T_n(x) = \cos(n \arccos x) \ (n \geq 0) \tag{5.26}$$

Although this appears at first glance to be a trigonometric function, using the trigonometric identity function in (5.26) leads to the following expressions for $T_n(x)$.

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\vdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \ (n \geq 1) \end{aligned}$$

The Chebyshev polynomials are orthogonal with weight $(1-x^2)^{-\frac{1}{2}}$ on the interval $[-1, 1]$. In particular, the following relationships hold.

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & (i \neq j) \\ \frac{\pi}{2} & (i = j \neq 0) \\ \pi & (i = j = 0) \end{cases} \tag{5.27}$$

The polynomial $T_n(x)$ has n zero points on the interval $[-1, 1]$. The positions of those zero points are

$$x = \cos\left[\frac{\pi(k-\frac{1}{2})}{n}\right] \ (k = 1, 2, \cdots, n) \tag{5.28}$$

In the same interval, $n+1$ extrema, which are at the following points,

$$x = \cos\left[\frac{\pi k}{n}\right] \ (k = 0, 1, \cdots, n)$$

appear alternately as relative maximum and relative minimum values. All relative maximum values are $T_n(x) = 1$, and all relative minimum values are $T_n(x) = -1$. This property of the Chebyshev polynomials helps reduce the error in the polynomial approximation of the function.

The Chebyshev polynomials satisfy discrete orthogonality relationships at the same time as the continuous orthogonality relationships of (5.27). That is, if x_k $(k = 1, \cdots, m)$ are m zero points of $T_m(x)$ given by (5.28), then the following holds if $i, j < m$

$$\sum_{k=1}^{m}(T_i(x_k)T_j(x_k)) = \begin{cases} 0 & (i \neq j) \\ \frac{m}{2} & (i = j \neq 0) \\ m & (i = j = 0) \end{cases} \tag{5.29}$$

The following property is obtained from the relationships in (5.26), (5.28) and (5.29).

When $f(x)$ is an arbitrary function defined on the interval $[-1, 1]$, if the $N+1$ coefficients c_j $(j = 0, 1, \cdots, N)$ are defined as follows for a sufficiently large N

$$\begin{aligned} c_j &= \frac{2}{N}\sum_{k=1}^{N}(f(x_k)T_j(x_k)) \\ &= \frac{2}{N}\sum_{k=1}^{N}(f\left(\cos\left[\frac{\pi(k-\frac{1}{2})}{N}\right]\right)\cos\left[\frac{\pi j(k-\frac{1}{2})}{N}\right]) \end{aligned} \tag{5.30}$$

then the following holds

$$f(x) \sim \left[\sum_{j=0}^{N}(c_j T_j(x))\right] - \frac{1}{2}c_0 \tag{5.31}$$

In particular, at all N given zero points of $T_N(x)$ (that is when $x = x_k$), the left side of (5.31) equals the right side.
For a fixed N, (5.31) is a polynomial of x, and that polynomial approximates the function $f(x)$ on the interval $[-1, 1]$ (which includes all zero points of $T_N(x)$).
By ignoring c_k $(k = m+1, \cdots, N)$, (5.31) can be reduced to the most accurate polynomial among the polynomials of the same degree, which is represented by the following expression.

$$f(x) \sim \left[\sum_{k=0}^{m}(c_k T_k(x))\right] - \frac{1}{2}c_0 \tag{5.32}$$

This is because the difference between (5.32) and (5.31) does not become greater than the sum of the ignored c_k $(k = m+1, \cdots, N)$ since $T_k(x)$ all take a value between ± 1. In particular, if we consider the case when the c_k quickly decrease, most of the error is due to the oscillatory function $c_{m+1}T_{m+1}(x)$ having $m+2$ equal extrema that are almost uniformly distributed on the interval $[-1, 1]$. This property, which is called extending the error uniformly, is important in approximating the function.
To generalize the expression in (5.30), perform the variable transformation

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \tag{5.33}$$

to let the domain of the function $f(x)$ being approximate be $[a, b]$. This transformation enables a function $f(x)$ on an arbitrary interval to be approximated by a Chebyshev polynomial of y $(-1 \leq y \leq 1)$.

(2) Polynomial approximation according to Chebyshev coefficients
Next, by transforming the Chebyshev coefficients c_k that were obtained to polynomial coefficients of the original variable x, we obtain the following approximation polynomial

$$f(x) \sim \sum_{k=0}^{m}(g_k x^k) \qquad a \leq x \leq b \tag{5.34}$$

However, although the coefficients g_k of expression (5.34) reflect the original Chebyshev approximation, calculating this expression requires a higher calculation precision than calculating the Chebyshev summation (5.32). This is because of the following reason.
For example, if we consider a Chebyshev polynomial of degree 30, although T_{30} is given as

$$T_{30}(x) = 2^{29}x^{30} + \cdots \tag{5.35}$$

it is actually expressed as $T_{30}(x) = \cos(30 \arccos(x))$. Since $-1 \leq T_{30}(x) \leq 1$, even though calculations involving large coefficients occurred, the value of T_{30} remained between ± 1. At this time, canceling occurred in the calculations involving large numbers, causing precision to drop. Therefore, the Chebyshev approximation should be represented as a polynomial only when the degree m of the Chebyshev approximation polynomial does not exceed 7 or 8. Note that even in this case, the precision decreases by two places in the

calculation in (5.35).
By performing the following procedure sequentially, the coefficients g_k can be derived from the transformed Chebyshev coefficients c_k $(k = 0, 1, \cdots, m)$ for a suitable value of m.
When the coefficients c_k $(k = 0, 1, \cdots, m)$ are given, obtain the coefficients d_k $(k = 0, 1, \cdots, m)$ so that the following relationships holds

$$\sum_{k=0}^{m}(d_k y^k) = \sum_{k=0}^{m}(c_k T_k(y))$$

Here, the following Clenshaw recurrence is used.

$$\begin{aligned} d_{m+2} &\equiv d_{m+1} \equiv 0 \\ d_j &= 2xd_{j+1} - d_{j+2} + c_j \ (j = m, m-1, \cdots, 1) \\ f(x) &\equiv d_0 = xd_1 - d_2 + \frac{1}{2}c_0 \end{aligned}$$

Next, when the coefficients d_k $(k = 0, 1, \cdots, m)$ are given, obtain the coefficients g_k $(k = 0, 1, \cdots, m)$ so that the following relationships holds

$$\sum_{k=0}^{m}(d_k y^k) = \sum_{k=0}^{m}(g_k x^k)$$

(Here, x and y are related according to (5.33). That is, the interval $-1 \leq y \leq 1$ is mapped to the interval $a \leq x \leq b$.)
First, perform the following transformation.

$$g_k = d_k \left(\frac{b-a}{2}\right)^k$$

Next, obtain g_k by synthetic division.
Here, let's try to obtain the value of $g_k(z)$ at $x = z$.
First, assume the polynomial to be obtained is

$$g_k(x) = \sum_{k=0}^{m}(g_k x^k) \tag{5.36}$$

If we assume that this has been divided by $(x - z)$, it can be written as

$$g_k(x) = (x - z)\sum_{k=0}^{m-1}(d_k x_{k-1}) \tag{5.37}$$

If we compare the coefficients of x_k in the two expressions above, we obtain

$$\begin{aligned} g_0 &= d_0 \\ g_k &= zg_{k-1} + d_k \ (k = 1, 2, \cdots, m) \end{aligned}$$

On the other hand, since the following clearly holds

$$g_m(z) = d_m$$

if we let $z = \dfrac{b+a}{2}$, then from (5.33), starting from $d_0 = g_0$, the values of g_k are obtained by the recurrence of (5.36) and (5.37).

5.1.3 Reference Bibliography

(1) Powell, M. J. D. , "A Hybrid Method for Nonlinear Equations", Numerical Methods for Nonlinear Algebraic Equations, P. Rabinowits, ed. , Gordon and Breach, pp.87-161, (1970).

(2) Ralston, A. and Rabinowitz, P. , "A First Course in Numerical Analysis", McGraw-Hill, Inc. , (1978).

(3) Balch, Stephen, J. and Thompson, Garth, T. , "An Efficient Algorithm For Polynomial Surface Fitting", Computers & Geosciences Vol.15, No.1, pp.107-119, (1989).

(4) William H. Press, "NUMERICAL RECIPES IN FORTRAN", CAMBRIDGE UNIVERSITY PRESS.

5.2 INTERPOLATION

5.2.1 ASL_dpdopl, ASL_rpdopl
Unequally Spaced Discrete Point Interpolation Value

(1) **Function**

ASL_dpdopl or ASL_rpdopl obtains the interpolation value f_l at the interpolation point x_l by using Aitken's scheme to interpolate n given points $(x_i, y_i)(i = 1, \cdots, n)$.

(2) **Usage**

Double precision:

```
ierr = ASL_dpdopl  (x, y, n, xl, eps, &fl, &dl, wk);
```

Single precision:

```
ierr = ASL_rpdopl  (x, y, n, xl, eps, &fl, &dl, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	X coordinates x_i of data points
2	y	{D∗ / R∗}	n	Input	Y coordinates y_i of data points
3	n	I	1	Input	Number of data points n
4	xl	{D / R}	1	Input	Interpolation point x_l
5	eps	{D / R}	1	Input	Required absolute precision (Default value: Unit for determining error×64) (See Section 5.1.2.5)
6	fl	{D∗ / R∗}	1	Output	Interpolation value f_l at x_l
7	dl	{D∗ / R∗}	1	Output	Absolute error of interpolation value (See Section 5.1.2.5)
8	wk	{D∗ / R∗}	n × 5	Work	Work area
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) eps $\geq$ Unit for determining error $\times 64$

(b) n ≥ 2

(c) x[i] $\neq$ x[j] (i $\neq$ j)

(d) $\min_{i=1,\cdots,n}(x[i-1]) \leq xl \leq \max_{i=1,\cdots,n}(x[i-1])$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) was not satisfied.	Processing is performed with the default value set.
2500	The precision could not be guaranteed since the absolute error of the interpolation value was not less than or equal to the required precision.	Processing is terminated without obtaining a solution having the required precision.
3000	Restriction (b) was not satisfied.	Processing is aborted.
3010	Restriction (c) was not satisfied.	
3020	Restriction (d) was not satisfied.	
4000	Overflow occurred during the interpolation value calculation.	

(6) **Notes**

(a) To obtain multiple interpolation values in the vicinity of x_l, use the function 5.2.2 $\left\{\begin{matrix}\text{ASL\_dpdapn}\\ \text{ASL\_rpdapn}\end{matrix}\right\}$ to obtain interpolation polynomial coefficients. Using that function will be more efficient than using this function. However, to obtain multiple interpolation values that are distant from x_l, using this function is more efficient since the value obtained by 5.2.2 $\left\{\begin{matrix}\text{ASL\_dpdapn}\\ \text{ASL\_rpdapn}\end{matrix}\right\}$ will deviate significantly from the true value due to the Runge phenomenon.

(7) **Example**

(a) Problem
Given the following input data:

i	x_i	y_i
1	0.0	0.0
2	60.0	0.0824
3	120.0	0.2747
4	180.0	0.6502

obtain the interpolation value at xl.

(b) Input data
Data points(x, y), n=4, xl=150.0 and
eps $= 5.0 \times 10^{-3}$.

(c) Main program

```
/*      C interface example for ASL_dpdopl */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int n;
    double x1;
    double eps;
    double y1;
    double d1;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dpdopl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dpdopl ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=4;
    x  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (n*5) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }
    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf%lf", &x[i],&y[i] );
    }

    fscanf( fp, "%lf", &x1 );
    fscanf( fp, "%lf", &eps);
    printf( "\tNumber of Data Points = %6d\n", n  );
    printf( "\n\tData Points (x,y)\n\n" );
    printf( "\t     i         x[i]        y[i]\n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t%6d    %8.3g    %8.3g\n", i,x[i],y[i] );
    }
    printf( "\t\n" );
    printf( "\tInterpolation Point = %8.3g \n",x1 );
    printf( "\tAbsolute Error      = %8.3g \n",eps );

    fclose( fp );

    ierr = ASL_dpdopl(x, y, n, x1, eps, &y1, &d1, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tAbsolute Error      = %8.3g\n",d1 );
    printf( "\tInterpolated Value  = %8.3g\n",y1 );

    free( x );
    free( y );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dpdopl ***

 ** Input **
```

```
Number of Data Points =      4

Data Points (x,y)

     i          x[i]         y[i]
     0            0            0
     1           60       0.0824
     2          120        0.275
     3          180         0.65

Interpolation Point =      150
Absolute Error      =    0.005

 ** Output **

ierr =      0

Absolute Error      =  0.00458
Interpolated Value  =    0.435
```

5.2.2 ASL_dpdapn, ASL_rpdapn
Unequally Spaced Discrete Point Interpolation Value and Interpolation Coefficients

(1) **Function**

ASL_dpdapn or ASL_rpdapn obtains the interpolation value f_l at the interpolation point x_l and the coefficients $c_i (i = 1, \cdots, m+1)$ of the interpolation polynomial

$$f(x) = c_1 + \sum_{i=2}^{m+1} \prod_{j=1}^{i-1} c_i (x - u_j)$$

by using Newton's method to interpolate n given points (x_i, y_i) $(i = 1, \cdots, n)$.

(2) **Usage**

Double precision:

```
ierr = ASL_dpdapn  (x, y, n, xl, m, c, &fl, u, wk);
```

Single precision:

```
ierr = ASL_rpdapn  (x, y, n, xl, m, c, &fl, u, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	X coordinates x_i of cross section lines
2	y	{D∗ / R∗}	n	Input	Y coordinates y_i of cross section lines
3	n	I	1	Input	Number of data points n
4	xl	{D / R}	1	Input	Interpolation point x_l
5	m	I	1	Input	Degree m of interpolation polynomial
6	c	{D∗ / R∗}	m×m+1	Output	Newton's divided differences (interpolation polynomial coefficients c_i = c[m ∗ (i − 1)] (See Notes (a))
7	fl	{D∗ / R∗}	1	Output	Interpolation value f_l at x_l
8	u	{D∗ / R∗}	n	Output	Sorted X coordinate values u_j
9	wk	{D∗ / R∗}	n × 2	Work	Work area
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $m > 0$

(b) $n \geq m + 1$

(c) $x[i] \neq x[j]$ $(i \neq j)$

(d) $\min_{i=1,\cdots,n}(x[i-1]) \leq xl \leq \max_{i=1,\cdots,n}(x[i-1])$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) or (b) was not satisfied.	Processing is aborted.
3010	Restriction (c) was not satisfied.	
3020	Restriction (d) was not satisfied.	
4000	Overflow occurred during the interpolation value calculation.	

(6) **Notes**

(a) The coefficients $c[m*(i-1)]$ $(i = 1, 2, \cdots, m+1)$ that are obtained are coefficients of the polynomial expressed as: $yl = c[0] + c[m] \times (xl - u[0])$
$+c[2*m] \times (xl - u[0]) \times (xl - u[1])$
$+\cdots$
$+c[m*m] \times (xl - u[0]) \times \cdots \times (xl - u[m-1])$
Therefore, to obtain the approximation value in the vicinity of xl by using these coefficients, perform the following calculation, for example.
Sample calculation (obtains the approximation value y at point x using c[m∗i])

```
y=c[m*m];
for(i=m-1; i>-1; i--){
  y=c[m*i]+(x-u[i])*y;
}
```

Although the method described above is efficient for obtaining an approximation value of a value in the vicinity of xl, if this method is used to obtain an approximation value of a value distant from xl, the precision may be poor. Therefore, this function should be used again or function 5.2.1 $\left\{\begin{matrix}\text{ASL\_dpdopl}\\ \text{ASL\_rpdopl}\end{matrix}\right\}$ should be used.

(b) In this function, only $m + 1$ input data x[i] in the vicinity of xl are efficient.

(7) **Example**

(a) Problem

Given the following input data:

i	x_i	y_i
1	−1.0	9.0
2	0.0	6.0
3	−3.0	−6.0
4	1.0	−4.0
5	2.0	−6.0

obtain the coefficients of the interpolation polynomial for these points and the interpolation value.

(b) Input data

Data points(x, y), n=5, xl=-2.0 and m=4.

(c) Main program

```
/*      C interface example for ASL_dpdapn */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int n;
    double x1;
    int m;
    double *c;
    double y1;
    double *sx;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dpdapn.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dpdapn ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=5;
    m=4;
    x  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }
    c  = ( double * )malloc((size_t)( sizeof(double) * (m*(m+1)) ));
    if( c  == NULL )
    {
        printf( "no enough memory for array c\n" );
        return -1;
    }
    sx  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( sx  == NULL )
    {
        printf( "no enough memory for array sx\n" );
        return -1;
    }
    wk  = ( double * )malloc((size_t)( sizeof(double) * (n*2) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }
    for( i=0 ; i<n ; i++ )
```

```
    {
        fscanf( fp, "%lf%lf", &x[i],&y[i] );
    }

    fscanf( fp, "%lf", &x1 );
    printf( "\n\tNumber of Data Points = %6d\n", n );
    printf( "\tInterpolating Polynomial's Degree = %6d\n", m );
    printf( "\n\tData Points (x,y)\n\n" );
    printf( "\t       i          x[i]         y[i]\n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t  %6d     %8.3g     %8.3g\n", i,x[i],y[i] );
    }
    printf( "\n\tInterpolation Point\n\n" );
    printf( "\t  %8.3g\n" ,x1);

    fclose( fp );

    ierr = ASL_dpdapn(x, y, n, x1, m, c, &y1, sx, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tSorted x\n\n" );
    for( i=0 ; i<m+1 ; i++ )
    {
        printf(  "\t  u[%6d] = %8.3g\n", i,sx[i] );
    }
    printf( "\n\tInterpolated Value\n\n" );
    printf( "\t  %8.3g\n" ,y1);
    printf( "\n\tCoefficient\n\n" );
    for( i=0 ; i<m+1 ; i++ )
    {
        printf(  "\t  c[%6d] = %8.3g\n", i,c[m*i] );
    }
    free( x   );
    free( y   );
    free( c   );
    free( sx  );
    free( wk  );

    return 0;
}
```

(d) Output results

```
*** ASL_dpdapn ***

 ** Input **


Number of Data Points =      5
Interpolating Polynomial's Degree =      4

Data Points (x,y)

       i          x[i]         y[i]
       0            -1           9
       1             0           6
       2            -3          -6
       3             1          -4
       4             2          -6

Interpolation Point

        -2

 ** Output **

ierr =      0

Sorted x

  u[     0] =       -1
  u[     1] =       -3
  u[     2] =        0
  u[     3] =        1
  u[     4] =        2

Interpolated Value

         2

Coefficient

  c[     0] =        9
  c[     1] =      7.5
  c[     2] =     -3.5
  c[     3] =        0
  c[     4] =      0.5
```

5.3 SURFACE INTERPOLATION

5.3.1 ASL_dplopl, ASL_rplopl
Discrete Point Interpolation Value on Two-Dimensional Cross Section Lines

(1) **Function**

ASL_dplopl or ASL_rplopl establishes straight lines (called cross section lines here) $x = x_i$ $(i = 1, 2, \cdots, nx)$ parallel to the Y axis in the X, Y plane and obtains the interpolation value f_l at an arbitrary point (x_l, y_l) by assigning data points $(y_{j,i}, z_{j,i})$ $(j = 1, \cdots, ny_i)$ on each of those x_i cross sections and interpolating these according to a cubic spline function. This function also obtains spline coefficients used for interpolating data on the given cross section lines.

(2) **Usage**

Double precision:

```
ierr = ASL_dplopl  (x, nx, y, z, my, ny, xl, yl, &fl, csp, &isw, wk);
```

Single precision:

```
ierr = ASL_rplopl  (x, nx, y, z, my, ny, xl, yl, &fl, csp, &isw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X Coordinates x_i of cross section lines
2	nx	I	1	Input	Number of cross section lines (dimension of array x) nx
3	y	{D∗ / R∗}	my×nx	Input	Y coordinate values $y_{i,j}$ of data points on cross section line x_i
4	z	{D∗ / R∗}	my×nx	Input	Z coordinate values $z_{i,j}$ of data points on cross section line x_i
5	my	I	1	Input	Maximum number of data points on cross section lines $\max(ny_i)$
6	ny	I*	nx	Input	Number of data points ny_i on each cross section line
7	xl	{D / R}	1	Input	Interpolation point X coordinate value x_l
8	yl	{D / R}	1	Input	Interpolation point Y coordinate value y_l
9	fl	{D∗ / R∗}	1	Output	Interpolation value f_l at interpolation point (x_l, y_l)
10	csp	{D∗ / R∗}	See Contents	Output	Spline coefficients for data on each cross section line **Size**: $3 \times (\text{my} - 1) \times \text{nx}$
11	isw	I*	1	Input/ Output	Processing switch. Enter 0 as the initial value. After processing terminates, 1 is returned. If isw $\neq 0$, processing is performed using the spline coefficients of the previous execution.
12	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: $5 \times \text{nx} - 3$
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $nx \geq 2,\ ny[i-1] \geq 2\ (i = 1, \cdots, nx)$

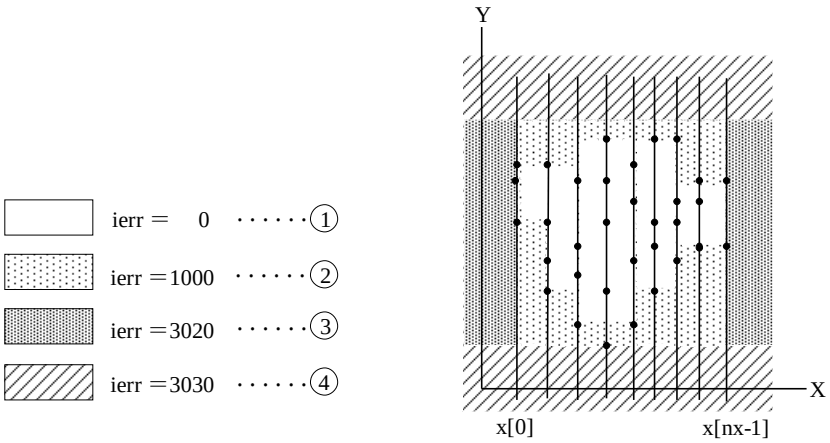
(b) $x[0] < x[1] < \cdots\cdots < x[nx-1]$ (ascending order)
$y[my*(i-1)] < y[1+my*(i-1)] < \cdots < y[ny[i-1]-1+my*(i-1)]$
$(i = 1, \cdots, nx)$ (ascending order)

(c) $x[0] \leq xl \leq x[nx-1]$

(d) $\min\limits_{1 \leq i \leq nx} y[j-1+my*(i-1)] \leq yl \leq \max\limits_{1 \leq i \leq nx} y[j-1+my*(i-1)]$

- Relationship between interpolation point and ierr
  The Figure 5−5 shows X-Y coordinates of data points. The vertical lines (on which black dots are displayed within the figure) indicate cross section lines. The ierr values are determined according to the position of the interpolation point (xl, yl). (Within the figure, the boundary is included in the area for which the value is smaller.)

Figure 5−5



(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The interpolation point is in the range of indicated by ② in Figure 5−5.	A value extrapolated by using the spline coefficients at the end points is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied. (The interpolation point is in the range indicated by ③ in Figure 5−5)	
3030	Restriction (d) was not satisfied. (The interpolation point is in the range indicated by ④ in Figure 5−5)	

(6) **Notes**

(a) The spline coefficients csp and the following function can be used to obtain the volume of the region that surrounds the entered data points:

6.2.7 {ASL_dgiipc / ASL_rgiipc}, 6.2.20 {ASL_dgiicz / ASL_rgiicz}

For: s···Area of cross section (Real; Size nx)
v···Volume of solid (Real; Size 1)
c···Work (Real; Size 3×(nx − 1))

the main portion of the program is as follows
(double precision example):

```
for(i=0; i<nx; i++)
{
   ierr = ASL_dgiicz(&y[my*i], &z[my*i], ny[i], *csp[3*i],
          y[my*i], y[ny[i]-1+my*i], &s[i]);
}
ierr = ASL_dgiipc(x, s, nx, x[0], x[nx-1], &v, c);
```

(b) The first time this function is called, set isw to 0. If you do not, the results will be invalid. For the second and subsequent time this function is called, when interpolating a different interpolation point using the same data, retain the contents of isw, set new values for xl and yl, and call this function again. (At this time, the value of isw has been changed to 1.) In this case, processing will be faster than the first time the function was called since processing is performed using spline coefficients that have already been obtained.

≀
xl=(Interpolation point X coordinate value ①)
yl=(Interpolation point Y coordinate value ①)
isw=0 ····· (Set isw)
ierr={ASL_dplopl / ASL_rplopl} (x, ···, xl, yl, ···, &isw, ···) ····· (First time)
≀
xl=(Interpolation point X coordinate value ②)
····· (Do not set isw)
yl=(Interpolation point Y coordinate value ②) ⋮
.....
ierr={ASL_dplopl / ASL_rplopl} (x, ···, xl, yl, ···, &isw, ···) ····· (Second time)
≀

(7) **Example**

(a) Problem

Given the following input data:

$$x_1 = 1.0$$
$$x_2 = 2.0$$
$$x_3 = 3.0$$
$$x_4 = 4.0$$
$$x_5 = 5.0$$

j	1	2	3	4	5	6
$y_{j,1}$ =	0.0,	1.0,	2.0,	3.0		
$z_{j,1}$ =	3.0,	2.82843,	2.23607,	0.0		
$y_{j,2}$ =	0.0,	1.0,	2.0,	3.0,	4.0	
$z_{j,2}$ =	4.0,	3.87298,	3.46410,	2.64575,	0.0	
$y_{j,3}$ =	0.0,	1.0,	2.0,	3.0,	4.0,	4.58258
$z_{j,3}$ =	4.58258,	4.47214,	4.12311,	3.46410,	2.23607,	0.0
$y_{j,4}$ =	0.0,	1.0,	2.0,	3.0,	4.0,	4.89898
$z_{j,4}$ =	4.89898,	4.79583,	4.47214,	3.87298,	2.82843,	0.0
$y_{j,5}$ =	0.0,	1.0,	2.0,	3.0,	4.0,	5.0
$z_{j,5}$ =	5.0,	4.89898,	4.58258,	4.0,	3.0,	0.0

obtain the interpolation value at (x_l, y_l) = (1.6, 2.3) and (3.2, 1.8).

(b) Input data

x, nx = 5, y, z, my = 6, xl[0] = 1.6, yl[0] = 2.3, xl[1] = 3.2,
yl[1] = 1.8, ny[0] = 4, ny[1] = 5, ny[2] = 6, ny[3] = 6 and
ny[4] = 6.

(c) Main program

```
/*      C interface example for ASL_dplopl */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    int nx;
    double *y;
    double *z;
    int my;
    int *ny;
    double xl1;
    double yl1;
    double xl2;
    double yl2;
    double fl1;
    double fl2;
    double *csp;
    int isw;
    double *wk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dplopl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dplopl ***\n" );
    printf( "\n     ** Input **\n\n" );
```

```
nx=5;
my=6;
x  = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x  == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}
ny  = ( int * )malloc((size_t)( sizeof(int) * nx ));
if( ny  == NULL )
{
    printf( "no enough memory for array ny\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * (my*nx) ));
if( y  == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}
z  = ( double * )malloc((size_t)( sizeof(double) * (my*nx) ));
if( z  == NULL )
{
    printf( "no enough memory for array z\n" );
    return -1;
}
csp  = ( double * )malloc((size_t)( sizeof(double) * (3*(my-1)*nx) ));
if( csp  == NULL )
{
    printf( "no enough memory for array csp\n" );
    return -1;
}
wk  = ( double * )malloc((size_t)( sizeof(double) * (5*nx-3) ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%d", &ny[i] );
}
for( i=0 ; i<nx ; i++ )
{
    for( j=0 ; j<ny[i] ; j++ )
    {
        fscanf( fp, "%lf", &y[j+my*i] );
    }
}
for( i=0 ; i<nx ; i++ )
{
    for( j=0 ; j<ny[i] ; j++ )
    {
        fscanf( fp, "%lf", &z[j+my*i] );
    }
}


fscanf( fp, "%lf", &xl1);
fscanf( fp, "%lf", &yl1);
fscanf( fp, "%lf", &xl2);
fscanf( fp, "%lf", &yl2);
printf( "\tnx    =   %6d\n", nx );
printf( "\tny[i] =   " );
for( i=0 ; i<nx ; i++ )
{
    printf(  "%6d ", ny[i] );
}
printf( "\n" );
printf( "\tmy    =   %6d\n", my );
printf( "\txl1   = %8.3g\n", xl1 );
printf( "\tyl1   = %8.3g\n", yl1 );
printf( "\txl2   = %8.3g\n", xl2 );
printf( "\tyl2   = %8.3g\n", yl2 );
printf( "\n\tx   =" );
for( i=0 ; i<nx ; i++ )
{
    printf(  "%8.3g ", x[i] );
}
printf( "\n" );
printf( "\n\ty   =\n" );
for( i=0 ; i<nx ; i++)
{
    printf( "\t     " );
```

```
        for( j=0 ; j<ny[i] ; j++ )
        {
            printf( "%8.3g ",  y[j+my*i] );
        }
        printf( "\n" );
    }
    printf( "\n\tz(x,y) =\n" );
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t     " );
        for( j=0 ; j<ny[i] ; j++ )
        {
            printf( "%8.3g ",  z[j+my*i] );
        }
        printf( "\n" );
    }

    fclose( fp );

    isw = 0;

    ierr = ASL_dplopl(x, nx, y, z, my, ny, xl1, yl1, &fl1, csp, &isw, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tInterpolated Value at (xl1,yl1)\n\n" );
    printf( "\t  fl1 = %8.3g\n" ,fl1);

    ierr = ASL_dplopl(x, nx, y, z, my, ny, xl2, yl2, &fl2, csp, &isw, wk);


    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tInterpolated Value at (xl2,yl2)\n\n" );
    printf( "\t  fl2 = %8.3g\n" ,fl2);

    free( x );
    free( y );
    free( z );
    free( ny );
    free( csp );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dplopl ***

 ** Input **

nx    =        5
ny[i] =        4      5      6      6      6
my    =        6
xl1   =      1.6
yl1   =      2.3
xl2   =      3.2
yl2   =      1.8

x   =       1        2        3        4        5

y   =
            0        1        2        3
            0        1        2        3        4
            0        1        2        3        4     4.58
            0        1        2        3        4      4.9
            0        1        2        3        4        5

z(x,y) =
            3     2.83     2.24        0
            4     3.87     3.46     2.65        0
         4.58     4.47     4.12     3.46     2.24        0
          4.9      4.8     4.47     3.87     2.83        0
            5      4.9     4.58        4        3        0

 ** Output **

ierr =      0

Interpolated Value at (xl1,yl1)

  fl1 =     2.85

 ** Output **

ierr =      0

Interpolated Value at (xl2,yl2)

  fl2 =     4.31
```

5.3.2 ASL_dpgopl, ASL_rpgopl
Discrete Point Interpolation Value on a Two-Dimensional Lattice

(1) **Function**

ASL_dpgopl or ASL_rpgopl obtains the interpolation value f_l at a single arbitrary point (x_l, y_l) within a lattice when all Z coordinate values $z_{i,j}$ on two-dimensional lattice points (x_i, y_i) $(i = 1, 2, \cdots, nx;\ j = 1, 2, \cdots, ny)$ are given.

(2) **Usage**

Double precision:

```
ierr = ASL_dpgopl  (x, nx, y, ny, z, eps, xl, yl, &fl, wk);
```

Single precision:

```
ierr = ASL_rpgopl  (x, nx, y, ny, z, eps, xl, yl, &fl, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values x_i of data points
2	nx	I	1	Input	Dimension nx of array x
3	y	{D∗ / R∗}	ny	Input	Y coordinate values y_i of data points
4	ny	I	1	Input	Dimension ny of array y
5	z	{D∗ / R∗}	nx×ny	Input	z[i − 1 + nx × (j − 1)] is the Z coordinate value $z_{i,j}$ at data point (x_i, y_j)
6	eps	{D / R}	1	Input	Required absolute precision (Default value: Unit for determining error × 64)
7	xl	{D / R}	1	Input	Interpolation point X coordinate value x_l
8	yl	{D / R}	1	Input	Interpolation point Y coordinate value y_l
9	fl	{D∗ / R∗}	1	Output	interpolation value f_l at interpolation point (x_l, y_l)
10	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: ny + 5 × max(nx, ny)
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) eps $\geq$ Unit for determining error $\times 64$

(b) nx ≥ 2, ny ≥ 2

(c) x[i $-$ 1] $\neq$ x[j $-$ 1], y[i $-$ 1] $\neq$ y[j $-$ 1] (i $\neq$ j)

(d) $\min\limits_{1\leq i\leq nx}(x[i-1]) \leq xl \leq \max\limits_{1\leq i\leq nx}(x[i-1])$
$\min\limits_{1\leq i\leq nx}(y[i-1]) \leq yl \leq \max\limits_{1\leq i\leq nx}(y[i-1])$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (a) was not satisfied.	Processing is performed with the default value set.
2500	The precision could not be guaranteed since the absolute error of the interpolation value was not less than or equal to the required precision.	Processing is terminated without obtaining a solution having the required precision.
3000	Restriction (b) was not satisfied.	Processing is aborted.
3010	Restriction (c) was not satisfied.	
3020	Restriction (d) was not satisfied.	
4000	Overflow occurred during the calculation.	

(6) **Notes**
None

(7) **Example**

(a) Problem
Given the following input data:

$x_1 = 1.0, \quad y_1 = 1.0$
$x_2 = 1.2, \quad y_2 = 1.2$
$x_3 = 1.4, \quad y_3 = 1.4$
$x_4 = 1.6, \quad y_4 = 1.6$
$\qquad\quad y_5 = 1.8$

$\xrightarrow{j}$

	$z_{i,j}$	y_1	y_2	y_3	y_4	y_5
	x_1	2.0,	1.56,	1.04,	0.44,	-0.24
$i\downarrow$	x_2	2.88,	2.44,	1.92,	1.32,	0.64
	x_3	3.92,	3.48,	2.96,	2.36,	1.68
	x_4	5.12,	4.68,	4.16,	3.56,	2.88

obtain the interpolation value at $(x_l, y_l) = (1.3, 1.5)$.

(b) Input data
x, nx=4, y, ny=5, z, eps=0.1, xl and yl.

(c) Main program

```
/*      C interface example for ASL_dpgopl */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    int nx;
    double *y;
    int ny;
    double *z;
    double eps;
    double xl;
    double yl;
    double fl;
    double *wk;
    int ierr;
    int i,j,nwk;
    FILE *fp;

    fp = fopen( "dpgopl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dpgopl ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=4;
    ny=5;
    x  = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * ny ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }
    z  = ( double * )malloc((size_t)( sizeof(double) * (nx*ny) ));
    if( z  == NULL )
    {
        printf( "no enough memory for array z\n" );
        return -1;
    }
    nwk=ny+5*( (nx>ny) ? nx : ny );
    wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
    }
    for( i=0 ; i<ny ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
    }
    for( i=0 ; i<nx ; i++ )
    {
        for( j=0 ; j<ny ; j++ )
        {
            fscanf( fp, "%lf", &z[i+nx*j] );
        }
    }


    fscanf( fp, "%lf", &eps);
    fscanf( fp, "%lf", &xl );
    fscanf( fp, "%lf", &yl );
    printf( "\tnx  =   %6d\n", nx );
    printf( "\tny  =   %6d\n", ny );
    printf( "\teps = %8.3g\n", eps );
    printf( "\txl  = %8.3g\n", xl  );
    printf( "\tyl  = %8.3g\n", yl  );
    printf( "\n\tData Points (x,y)\n\n" );
    printf( "\t  x = " );
    for( i=0 ; i<nx ; i++ )
    {
```

```
        printf(  "%8.3g ", x[i] );
    }
    printf( "\n\n" );
    printf( "\t  y =\n" );
    printf( "\t      " );
    for( i=0 ; i<ny ; i++ )
    {
        printf(  "%8.3g ", y[i] );
    }
    printf( "\n\n" );
    printf( "\t  z(x,y) =\n" );
    for( i=0 ; i<nx  ; i++ )
    {
        printf( "\t      " );
        for( j=0 ; j<ny ; j++ )
        {
            printf( "%8.3g ",  z[i+nx*j] );
        }
        printf( "\n" );
    }

    fclose( fp );

    ierr = ASL_dpgopl(x, nx, y, ny, z, eps, xl, yl, &fl, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tInterpolated Value at (xl,yl)\n\n" );
    printf( "\t  %8.3g\n" ,fl );

    free( x );
    free( y );
    free( z );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dpgopl ***

 ** Input **

nx  =        4
ny  =        5
eps =      0.1
xl  =      1.3
yl  =      1.5

Data Points (x,y)

  x =        1      1.2      1.4      1.6

  y =
             1      1.2      1.4      1.6      1.8

  z(x,y) =
             2     1.56     1.04     0.44    -0.24
          2.88     2.44     1.92     1.32     0.64
          3.92     3.48     2.96     2.36     1.68
          5.12     4.68     4.16     3.56     2.88

 ** Output **

ierr =      0

Interpolated Value at (xl,yl)

      2.13
```

5.4 LEAST SQUARES APPROXIMATION

5.4.1 ASL_dndaao, ASL_rndaao
Least Squares Approximation Orthogonal Polynomial Having Automatically Determined Degree

(1) **Function**

ASL_dndaao or ASL_rndaao takes (x_i, y_i) $(i = 1, \cdots, n)$ as given coordinate values and obtains the degree m, coefficients a_j $(j = 1, \cdots, m+1)$, and $f(x_i)$ values of the optimum polynomial $f(x) = \sum_{j=1}^{m+1} a_j x^{m+1-j}$ that minimizes the weighted sum of the squares of the differences of the approximate value $f(x_i)$ and y_i, which is expressed as $\sum_{i=1}^{n} w(x_i)\{y_i - f(x_i)\}^2$.

(2) **Usage**

Double precision:

```
ierr = ASL_dndaao  (x, y, w, n, a, &m, &sx, f, wk);
```

Single precision:

```
ierr = ASL_rndaao  (x, y, w, n, a, &m, &sx, f, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	X coordinates x_i of data points
2	y	{D∗ / R∗}	n	Input	Y coordinates y_i of data points
3	w	{D∗ / R∗}	n	Input	Weight function values w[i − 1] = w(x$_i$) at data points (See Notes (a))
4	n	I	1	Input	Number of data points n
5	a	{D∗ / R∗}	m + 1	Output	Coefficients a_i of approximation polynomial (a[0]: Coefficient of highest degree term, ⋯, a[m]: Constant term)
6	m	I*	1	Output	Degree m of optimum approximation polynomial
7	sx	{D∗ / R∗}	1	Output	If ierr = 0, sx = 0 If ierr = 1000, a is the array of polynomial coefficients for (x + sx)
8	f	{D∗ / R∗}	n	Output	Approximate value of $f(x_i)$ of Y coordinate at x_i
9	wk	{D∗ / R∗}	n × 8	Work	Work area
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(b) w[i − 1] ≥ 0 (i = 1, ⋯, n)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Since the X coordinate range does not include the origin, the approximation values are not accurately obtained by using the general polynomial coefficients.	Polynomial coefficients for $f(x+s)$ are output to a and s is output to sx. (See Notes (b))
3000	Restriction (a) or (b) was not satisfied.	Processing is aborted.
3010	The number of data points for which the weight is not 0.0 and the X coordinates are considered to differ is less than the optimum degree plus one. (X coordinates are considered to differ here if the difference of the X coordinate at one point with that at another point is greater than the entire range times the square root of the unit for determining error.	
4000	Overflow or division by zero occurred during the polynomial coefficient calculation.	

(6) **Notes**

(a) If $w(x_k)$ is greater than $w(x_j)$ $(j \neq k)$, then the value of f at $x = x_k$ is closer to the value of y_k than the value of f at $x = x_j$ is to the value of y_j.

(b) The origin at which $x = 0$ must be included in the X, Y coordinate distribution range. Since the precision of the approximation value calculations by a polynomial in terms of x will decrease significantly if the origin is not included, the following processing is performed.

- ierr = 1000 is returned.
- The amount s that the x coordinates are shifted is returned in sx.
- The coefficients of the polynomial for $f(x+s) = \sum_{i=1}^{m+1} a'_i (x+s)^{m+1-i}$ are returned in a.

Therefore, the approximation polynomial value calculation will be, for example, as follows.
Sample calculation (obtains the approximation polynomial value y at point x)

```
if (ierr==1000){
  x+=sx;
}
y = a[0];
for (i=1; i<m+1; i++){
  y = y*x+a[i];
}
```

However, the y value obtained by this calculation for input data point x may differ in the error range from the f value obtained by using the function. The reason is that the f value is obtained by using an intermediate orthogonal polynomial that is used for obtaining the approximation polynomial within the function.

(7) **Example**

(a) Problem

Given the following input data:

i	x_i	y_i	$w(x_i)$
1	0.0	5.312	1.0
2	0.3	6.044	1.0
3	0.7	8.276	1.0
4	1.0	11.000	1.0
5	1.2	13.496	1.0

obtain the coefficients of the approximation polynomial.

(b) Input data

Data points (x[i], y[i]), weight function values w[i] and n = 5.

(c) Main program

```
/*      C interface example for ASL_dndaao */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    double *w;
    int n;
    double *a;
    int m;
    double se;
    double *f;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dndaao.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dndaao ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=5;
    x  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }
    w  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( w  == NULL )
    {
        printf( "no enough memory for array w\n" );
        return -1;
    }
    a  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( a  == NULL )
    {
        printf( "no enough memory for array a\n" );
        return -1;
    }
    f  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( f  == NULL )
    {
        printf( "no enough memory for array f\n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (n*8) ));
```

```
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf %lf %lf", &x[i],&y[i],&w[i] );
    }
    printf( "\tNumber of Data Points =%6d\n", n );
    printf( "\n\tData Points (x,y) ,Weight Function Value \n\n" );
    printf( "\t       i           x[i]          y[i]          w[i] \n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t  %6d      %8.3g      %8.3g      %8.3g\n", i,x[i],y[i],w[i] );
    }

    fclose( fp );

    ierr = ASL_dndaao(x, y, w, n, a, &m, &se, f, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tOptimal Degree = %6d\n", m );
    printf( "\n\tsx = %8.3g\n", se );
    printf( "\n\tCoefficients of Polynomial\n\n" );
    for( i=0 ; i<m+1 ; i++ )
    {
        printf( "\t  a[%6d]=%8.3g\n", i,a[i] );
    }
    printf( "\n\tApproximate Value\n\n" );
    for( i=0 ; i<n ; i++ )
    {
        printf( "\t  f[%6d]=%8.3g\n", i,f[i] );
    }

    free( x );
    free( y );
    free( w );
    free( a );
    free( f );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dndaao ***

 ** Input **

Number of Data Points =     5

Data Points (x,y) ,Weight Function Value

       i           x[i]          y[i]          w[i]
       0             0          5.31             1
       1           0.3          6.04             1
       2           0.7          8.28             1
       3             1            11             1
       4           1.2          13.5             1

 ** Output **

ierr =      0

Optimal Degree =      4

sx =        0

Coefficients of Polynomial

  a[     0]=    1.24
  a[     1]=   -1.96
  a[     2]=    5.47
  a[     3]=   0.942
  a[     4]=    5.31

Approximate Value

  f[     0]=    5.31
  f[     1]=    6.04
  f[     2]=    8.28
  f[     3]=      11
  f[     4]=    13.5
```

5.4.2 ASL_dndapo, ASL_rndapo
Least Squares Approximation Orthogonal Polynomials

(1) **Function**

ASL_dndapo or ASL_rndapo takes (x_i, y_i) $(i = 1, \cdots, n)$ as given coordinate values and obtains the coefficients a_j $(j = 1, \cdots, m+1)$ and $f(x_i)$ values of the optimum m-th degree polynomial $f(x) = \sum_{j=1}^{m+1} a_j x^{m+1-j}$ that minimizes the weighted sum of the squares of the differences of the approximate value $f(x_i)$ and y_i, which is expressed as $\sum_{i=1}^{n} w(x_i)\{y_i - f(x_i)\}^2$.

(2) **Usage**

Double precision:

```
ierr = ASL_dndapo  (x, y, w, n, a, m, &sx, f, wk);
```

Single precision:

```
ierr = ASL_rndapo  (x, y, w, n, a, m, &sx, f, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	X coordinates x_i of data points
2	y	{D∗ / R∗}	n	Input	Y coordinates y_i of data points
3	w	{D∗ / R∗}	n	Input	Weight function values w[i−1] = w(x$_i$) of data points (See Notes (a))
4	n	I	1	Input	Number of data points
5	a	{D∗ / R∗}	m + 1	Output	Coefficients a_i of approximate polynomial (a[0]: Coefficient of highest degree term, ⋯, a[m]: Constant term)
6	m	I	1	Input	Degree of approximate polynomial
7	sx	{D∗ / R∗}	1	Output	If ierr = 0, sx = 0 If ierr = 1000, a is the array of polynomial coefficients for (x+sx)

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	f	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	n	Output	Approximate value $f(x_i)$ of Y at X
9	wk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	$\text{n} \times 8$	Work	Work area
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) m>0

(b) $\text{n}' \geq \text{m} + 1$ (n′:Number of data points for which the weight is not 0.0 and the X coordinate spread exceeds $\sqrt{\varepsilon}$ of the entire data distribution range) (ε: Unit for determining error)

(c) $\text{w}[\text{i} - 1] \geq 0$ $(\text{i} = 1, \cdots, \text{n})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Since the X coordinate range does not include the origin, approximation values are not accurately obtained by using the general polynomial coefficients.	Polynomial coefficients for $f(x + s)$ are output in a and s is output in sx. (See Notes (c))
3000	Restriction (a) or (c) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
4000	Overflow occurred during the calculation.	

(6) **Notes**

(a) If $w(x_k)$ is larger than $w(x_j)$ $(j \neq k)$, then the value of f at $x = x_k$ is closer to the value of y_k than the value of f at $x = x_j$ is to the value of y_j.

(b) If data is input for which the spread of the X coordinate values does not exceed $\sqrt{\varepsilon}$ of the X coordinate data distribution range, then the data must satisfy restriction (b).

(c) The origin at which $x = 0$ must be included in the X, Y coordinate distribution range. Since the precision of approximation value calculations by a polynomial in terms of x will decrease significantly if the origin is not included, the following processing is performed.

- ierr = 1000 is returned.
- The amount s that the X coordinates are shifted is returned in sx.
- The coefficients a'_i of the polynomial for $f(x + s) = \sum_{i=1}^{m+1} a'_i (x + s)^{m+1-i}$ are returned in a.

Therefore, the approximation polynomial value calculation will be, for example, as follows.

Sample calculation (obtains the approximation polynomial value y at point x)

```
if (ierr==1000){
  x+=sx;
}
```

```
y = a[0];
for (i=1; i<m+1; i++){
  y = y*x+a[i];
}
```

However, the y value obtained by this calculation for input data point x may differ in the error range from the f value obtained by using the function. The reason is that the f value is obtained by using an intermediate orthogonal polynomial that is used for obtaining the approximation polynomial within the function.

(d) Since a decrease in precision occurs more readily if a larger value is taken for the degree m of the approximate polynomial, double precision should be used.

(7) **Example**

(a) Problem
Given the following input data:

i	x_i	y_i	$w(x_i)$
1	0.0	5.312	1.0
2	0.3	6.044	1.0
3	0.7	8.276	1.0
4	1.0	11.000	1.0
5	1.2	13.496	1.0

obtain the coefficients of the approximate polynomial.

(b) Input data
Data points (x_i, y_i), weight function values $w(x_i)$, n=5 and m=3.

(c) Main program

```
/*      C interface example for ASL_dndapo */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    double *w;
    int n;
    double *a;
    int m;
    double se;
    double *f;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dndapo.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dndapo ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=5;
    m=3;
    x  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y  == NULL )
```

```
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }
    w  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( w  == NULL )
    {
        printf( "no enough memory for array w\n" );
        return -1;
    }
    a  = ( double * )malloc((size_t)( sizeof(double) * (m+1)  ));
    if( a  == NULL )
    {
        printf( "no enough memory for array a\n" );
        return -1;
    }
    f  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( f  == NULL )
    {
        printf( "no enough memory for array f\n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (n*8) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf,%lf,%lf", &x[i],&y[i],&w[i] );
    }
    printf( "\tDegree of Approximate Value =%6d\n", m );
    printf( "\tNumber of Data Points =%6d\n", n );
    printf( "\n\tData Points (x,y) ,Weight Function Value \n\n" );
    printf( "\t       i           x[i]          y[i]          w[i] \n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t  %6d      %8.3g      %8.3g      %8.3g\n", i,x[i],y[i],w[i] );
    }

    fclose( fp );

    ierr = ASL_dndapo(x, y, w, n, a, m, &se, f, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tsx = %8.3g\n", se );
    printf( "\n\tCoefficients of x\n\n" );
    for( i=0 ; i<m+1 ; i++ )
    {
        printf(  "\t  a[%6d]=%8.3g\n", i,a[i] );
    }
    printf( "\n\tApproximate Value\n\n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t  f[%6d]=%8.3g\n", i,f[i] );
    }

    free( x );
    free( y );
    free( w );
    free( a );
    free( f );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dndapo ***

 ** Input **

Degree of Approximate Value =     3
Number of Data Points =     5

Data Points (x,y) ,Weight Function Value

       i           x[i]          y[i]          w[i]
       0             0          5.31             1
       1           0.3          6.04             1
       2           0.7          8.28             1
       3             1            11             1
       4           1.2          13.5             1

 ** Output **
```

```
ierr =      0

sx =        0

Coefficients of x

  a[     0]=    1.02
  a[     1]=    3.28
  a[     2]=    1.41
  a[     3]=    5.31

Approximate Value

  f[     0]=    5.31
  f[     1]=    6.06
  f[     2]=    8.26
  f[     3]=      11
  f[     4]=    13.5
```

5.4.3 ASL_dndanl, ASL_rndanl
Least Squares Approximation Nonlinear Functions

(1) **Function**

ASL_dndanl or ASL_rndanl fits an approximate function $f(x_i, \boldsymbol{a})$ to given coordinate values (x_i, y_i) $(i = 1, \cdots, n)$ and optimizes parameters a_i $(i = 1, \cdots, m)$ where a_i are components of $\boldsymbol{a}$ so that the sum of the squares of the residuals $y_i - (x_i, \boldsymbol{a})$ is minimized.

(2) **Usage**

Double precision:

```
ierr = ASL_dndanl  (f, x, y, n, er, &nev, a, m, yf, &s, iwk, wk);
```

Single precision:

```
ierr = ASL_rndanl  (f, x, y, n, er, &nev, a, m, yf, &s, iwk, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	-	Input	Name of function f(x, a) of approximation function $f(x, \boldsymbol{a})$. (See Notes (a))
2	x	{D∗ / R∗}	n	Input	X coordinate values of data points x_i
3	y	{D∗ / R∗}	n	Input	Y coordinate values of data points y_i
4	n	I	1	Input	Number of data points n
5	er	{D / R}	1	Input	Required precision (Default value: $2 \times \sqrt{(\text{Unitfordeterminingerror})}$)
6	nev	I*	1	Input	Maximum number of evaluations of function $f(x, \boldsymbol{a})$ (Default value: 100 × n × m)
				Output	Actual number of function evaluations
7	a	{D∗ / R∗}	m	Input	Coefficient initial values $\boldsymbol{a}_0$
				Output	Optimum coefficients $\boldsymbol{a}^*$
8	m	I	1	Input	Number of coefficients m
9	yf	{D∗ / R∗}	n	Output	Value of approximate function $f(x_i, \boldsymbol{a}^*)$ at x_i
10	s	{D∗ / R∗}	1	Output	Minimum sum of squares value $s = \sum_{i=1}^{n} (y_i - f(x_i, \boldsymbol{a}^*))^2$
11	iwk	I*	4 × m	Work	Work area
12	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: n × (2 × m + 1) + m × (m + 4)
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $0 < \text{m} \leq \text{n}$

(b) er ≥ Unit for determining error (except when 0.0 is entered in order to set er to the default value)

(c) nev > 0 (except when 0 is entered in order to set nev to the default value)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Restriction (b) or (c) was not satisfied.	Processing is performed with the default value set for nev or er.
3000	Restriction (a) was not satisfied.	Processing is aborted.
4000	The linear least squares method could not be solved.	The values of a, yf, and s at that time are output and processing is aborted.
4100	The steepest descent could not be calculated.	
4200	The solution could not be corrected 2m times consecutively.	
5000	The values did not converge before the given maximum number of evaluations was reached.	

(6) **Notes**

(a) **Function**f should be created as follows.

```
double FORTRAN f(double *x, double *a) {
        return f(*x, a);
}
```

(b) Convergence is determined according to the following condition, and the solution is assumed to be $\boldsymbol{a} + \Delta\boldsymbol{a}$:

$$\|\Delta\boldsymbol{a}\| \leq \mathrm{er} \times \max(1, \|\boldsymbol{a} + \Delta\boldsymbol{a}\|)$$

where $\Delta\boldsymbol{a}$ is the correction vector for $\boldsymbol{a}$ and $\|\boldsymbol{a}\| = \max|a_i|$. A value on the order of the default value should be taken for er.

(c) If a default value is shown for an argument in the Contents column of the table in the argument section, then the default value will be set if 0 is entered for an integer-type argument or if 0.0 is entered for a real-type argument.

(d) The x_i and y_i coordinate values must be of nearly equal order. If the order differs, then the a_i values are obtained by scaling the x_i values so that they are on the same order as y_i, and then the a_i are transformed so that they become a_i for the original x_i values.

Example: Obtain the optimum a_1, a_2, a_3, a_4 for the function:

$$y = a_i[1 - \exp\{-(x - a_3)/a_2\}] + a_4$$

to fit a Weibull distribution curve.

(i) Assume:

$$y_{\min} = \min_{i=1,\cdots,n} y_i$$
$$x_{\min} = \min_{i=1,\cdots,n} x_i$$

Let:

$$y'_i = y_i - y_{\min}\ (i = 1, \cdots, n),\ a'_4 = y_{\min}$$
$$x'_i = x_i - x_{\min}\ (i = 1, \cdots, n),\ a'_3 = x_{\min}$$

Now, let:

$$y_{\max} = \max_{i=1,\cdots,n} y'_i$$
$$x_{\max} = \max_{i=1,\cdots,n} x'_i$$

(If $a_3 = a_4 = 0$, these operations are unnecessary.)

(These above operations perform a coordinate transformation.)

(ii) Let $s = y_{\max}/x_{\max}$ and set as follows:

$$x''_i = sx'_i (i = 1, \cdots, m) \text{(Scaling)}$$

(iii) Let the initial values of a_1 and a_2 be on the order of $y_{\max}$ and let the initial values of a_3 and a_4 be zero. Perform a nonlinear least squares approximation using data (x''_i, y'_i) $(i = 1, \cdots, n)$, for $f(x, \boldsymbol{a}) = a'_1[1 - \exp\{-(x - a'_3)/a'_2\}] + a'_4$, and obtain coefficients a'_1, a'_2, a'_3, and a'_4.

(iv) The coefficients a_2, a_3 and a_4 are as follows:

$$a_1 = a'_1$$
$$a_2 = a'_2/s$$
$$a_3 = a'_3/s + x_{\min}$$
$$a_4 = a'_4 + y_{\min}$$

(7) **Example**

(a) Problem

Obtain the optimum values of x_0, w, h, a_0, and a_1 by fitting the function:

```
(  −5.0,  2.7  )
(  −4.0,  2.9  )
(  −3.0,  3.1  )
(  −2.0,  3.4  )
(   1.0,  3.9  )
(   0.0,  4.7  )
(   1.0,  6.0  )
(   2.0,  7.8  )
(   3.0,  7.9  )
(   4.0,  6.3  )
(   5.0,  5.2  )
```

to the following 11 data points(x, y):

$$f(x) = \frac{hw^2}{(x - x_0)^2 + w^2} + a_0 + a_1 x$$

Assume the following initial values:

$x_0 = 0.0$, $w = 1.0$, $h = 6.0$, $a_0 = 3.5$ and $a_1 = 0.2$.

(b) Input data

Function name: fndanl.

Array x and y, n=11, er=0.0 and nev=0.

Array a:

Assign x_0, w, h, a_0, and a_1 sequentially to a[0], a[1], a[2], a[3] and a[4].

m=5.

(c) Main program

```
/*      C interface example for ASL_dndanl */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x,double *a)
#else
double f(x,a)
double *x;
double *a;
#endif
{
    double f1,f2;
    f1=a[2]*a[1]*a[1]/(((*x)-a[0])*((*x)-a[0])+(a[1]*a[1]));
    f2=a[3]+a[4]*(*x);
    return (f1+f2);
}
#ifdef __cplusplus
}
#endif

int main()
{
    double *xd;
    double *yd;
    int n;
    double er;
    int nev;
    double *x;
    int m;
    double *y;
    double s;
    int *iwk;
    double *wk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dndanl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dndanl ***\n" );
    printf( "\n     ** Input **\n\n" );
    n=11;
    m=5;
    xd  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( xd  == NULL )
    {
        printf( "no enough memory for array xd\n" );
        return -1;
    }
    yd  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( yd  == NULL )
    {
        printf( "no enough memory for array yd\n" );
        return -1;
    }
    x  = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * n ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    wk  = ( double * )malloc((size_t)( sizeof(double) * (n*(2*m+1)+m*(m+4)) ));
    if( wk  == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    iwk  = ( int * )malloc((size_t)( sizeof(int) * (3*m) ));
    if( iwk  == NULL )
    {
        printf( "no enough memory for array iwk\n" );
```

```
        return -1;
    }

    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf", &xd[i] );
    }
    for( i=0 ; i<n ; i++ )
    {
        fscanf( fp, "%lf", &yd[i] );
    }
    fscanf( fp, "%d", &nev );
    fscanf( fp, "%lf", &er );
    for( i=0 ; i<m ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
    }
    printf( "\tm  =  %6d\n", m );
    printf( "\tn  =  %6d\n", n );
    printf( "\tnev=  %6d\n", nev );
    printf( "\ter =%8.3g\n", er );
    printf( "\n\tCoordinates (x,y)\n\n" );
    printf( "\t       i           x[i]          y[i]\n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t  %6d      %8.3g      %8.3g\n", i,xd[i],yd[i] );
    }
    printf( "\n\tInitial Value of Coefficients\n\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf(  "\t  a[%6d]=%8.3g\n", i,x[i] );
    }

    fclose( fp );

    ierr = ASL_dndanl(f, xd, yd, n, er, &nev, x, m, y, &s, iwk, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n\tnev  = %6d\n", nev  );
    printf( "\n\tOptimized Coefficients\n\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf(  "\t  a[%6d]=%8.3g\n", i,x[i] );
    }
    printf( "\n\tLeast Squares\n\n" );
    printf( "\t  s =%8.3g\n",s );
    printf( "\n\tFunction Value\n\n" );
    for( i=0 ; i<n ; i++ )
    {
        printf(  "\t  yf[%6d]=%8.3g\n", i,y[i] );
    }

    free( xd );
    free( yd );
    free( x );
    free( y );
    free( iwk );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dndanl ***

 ** Input **

m  =       5
n  =      11
nev=       0
er =       0

Coordinates (x,y)

       i           x[i]          y[i]
       0            -5           2.7
       1            -4           2.9
       2            -3           3.1
       3            -2           3.4
       4            -1           3.9
       5             0           4.7
       6             1             6
       7             2           7.8
       8             3           7.9
       9             4           6.3
      10             5           5.2
```

```
Initial Value of Coefficients

  a[     0]=       0
  a[     1]=       1
  a[     2]=       6
  a[     3]=     3.5
  a[     4]=     0.2

 ** Output **

ierr =      0

nev  =    759

Optimized Coefficients

  a[     0]=    2.49
  a[     1]=     1.8
  a[     2]=    4.97
  a[     3]=    2.96
  a[     4]=   0.108

Least Squares

  s = 0.00361

Function Value

  yf[     0]=    2.69
  yf[     1]=    2.88
  yf[     2]=    3.12
  yf[     3]=    3.43
  yf[     4]=     3.9
  yf[     5]=    4.66
  yf[     6]=    6.02
  yf[     7]=     7.8
  yf[     8]=    7.89
  yf[     9]=    6.31
  yf[    10]=    5.19
```

5.5 LEAST SQUARES SURFACE APPROXIMATION

5.5.1 ASL_dnrapl, ASL_rnrapl
Two-Dimensional Arbitrary Data Least Squares Approximation Polynomial

(1) **Function**

ASL_dnrapl or ASL_rnrapl takes arbitrary data points on a plane $(x_k, y_k, z_k)(k = 1, \cdots, n)$ and obtains the coefficients $a_{i,j}$ and the approximate values $f(x_k, y_k)$ $(k = 1, \cdots, n)$ at the input data points of the m-th degree approximation polynomial for x and y:

$$f(x, y) = \sum_{j=1}^{m+1} \sum_{i=1}^{m+2-j} a_{i,j} x^{i-1} y^{j-1}$$

so that the sum of the squares of the differences of the polynomial values $f(x_k, y_k)$ and the Z coordinate values z_k at the data points is minimized.

(2) **Usage**

Double precision:

```
ierr = ASL_dnrapl  (x, y, z, n, m, a, f, iw, wk);
```

Single precision:

```
ierr = ASL_rnrapl  (x, y, z, n, m, a, f, iw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	X coordinates x_k of data points
2	y	{D∗ / R∗}	n	Input	Y coordinates y_k of data points
3	z	{D∗ / R∗}	n	Input	Z coordinates z_k of data points
4	n	I	1	Input	Number of data points
5	m	I	1	Input	Degree m for x and y of the approximation polynomial
6	a	{D∗ / R∗}	See Contents	Output	Coefficients $a_{i,j}$ of approximation polynomial **Size**: $(m+1) \times (m+2)/2$
7	f	{D∗ / R∗}	n	Output	Approximate value $f(x_k, y_k)$ of Z coordinate at (x_k, y_k)
8	iw	I*	See Contents	Work	Work area **Size**: $(m+1) \times (m+2)/2$
9	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: $(m+2)^4/4$
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq (m+1) \times (m+2)/2$

(b) $m \geq 0$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The degree of the approximation exceeded 10. (The approximation error may be large. (See Notes (c))	Processing continues using the assigned degree.
2100	There existed the diagonal element which was close to zero in the *LU* decomposition. The precise solutions or the inverse matrix may not be obtained.	Processing continues.
3000	Restriction (a) was not satisfied, or $n' < n^{'} < (m+1) \times (m+2)/2$ was satisfied where n' is the number of data points for which (x[k], y[k]) differed.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
4000+i	The pivot became 0.0 during the processing of the i-th step of the LU decomposition function used to solve the normal equation internally.	

(6) **Notes**

(a) If all input data points (x_k, y_k) $(k = 1, \cdots, n)$ lie on a single straight line, they cannot define a surface. The result obtained in this case is a single indefinite solution.

(b) In this function, the polynomial coefficients $a_{i,j}$ is stored in the one-dimensional array a to increase the efficiency. The correspondence relationship between $a_{i,j}$ and a is as follows:

$$a_{i,j} = \mathrm{a}[\mathrm{i} - 1 + \{(\mathrm{j} - 1) \times (2 \times \mathrm{m} - \mathrm{j} + 4)\}/2]$$

This corresponds as follows in the figure below. First, look at the terms listed in the first row from left to right. The first term, which is the constant term, corresponds to a[0]. Then, sequentially, a[1] =coefficient of x, a[2] = coefficient of $x^2, \cdots,$ a[m] = coefficient of x^m. Since we have reached the right end of the first row, move to the next row and look at the terms in a similar manner. a[m + 1] = coefficient of $y, \cdots,$ a[2m] = coefficient of yx^{m-1}. Proceeding in this way, terms are associated up until the $(m+1)$th row in which a$[-1 + (\mathrm{m} + 1)] \times (\mathrm{m} + 2)/2$ = coefficient of y^m.

$$\begin{array}{lllll} 1 & x & x^2 & x^3 & \cdots \quad x^m \\ y & xy & x^2y & \cdots & x^{m-1}y \\ y^2 & xy^2 & \cdots & \cdot & \\ \cdots & & \cdot & & \\ \cdots & \cdot & & & \\ y^m & & & & \end{array}$$

(c) Since this function approximates the given points on the surface by only one polynomial, meaningless results may be produced depending on the data. In particular, if the degree of the approximation polynomial is large, you should check whether or not the output results are appropriate. The fitting rate, which is defined by the following equation, is one characteristic for checking whether or not the

output results are appropriate. The fitting rate can be calculated by using the following program (double precision example, main portion only).

zs $\cdots$ $\| f_k \|_2 = \sqrt{\sum_{k=1}^{n-1} z[k]^2}$ (Real; Size: 1)

rs $\cdots$ $| x | = \sqrt{\sum_{k=1}^{n-1} (f[k] - z[k])^2}$ (Real; Size: 1) df $\cdots$ Work variable (Real; Size: 1)

fit $\cdots$ fitting Fitting rate $= \dfrac{\| f_k \|_2}{| x | + \| f_k \|_2} \times 100$ (Real; Size: 1)

The fitting rate can be calculated by using the follow program (main portion only).

```
zs = 0.0;
rs = 0.0;
for(k = 0; k < n; k++)
{
    zs = zs + z[k] * z[k];
    df = f[k] - z[k];
    rs = rs + df * df;
}
zs = sqrt(zs);
rs = sqrt(rs);
fit = 1.0e2;
if((zs != 0.0) || (rs != 0.0))
{
    fit = zs/(zs + rs) * 1.0e2;
}
```

If the fitting rate is low, use this function after subdividing the approximation interval and shifting or scaling the input data to the vicinity of the origin.

(d) In the following cases, the approximation polynomial coefficient values a may differ depending on the operating system or whether the calculations are single precision or double precision.

- The degree is high
- The input data values oscillate severely

(e) Method of using the polynomial coefficients a to obtain an approximate value at a point other than the input data points. For:

xl $\cdots$ x coordinate value of approximation point	(Real; Size: 1)
yl $\cdots$ y coordinate value of approximation point	(Real; Size: 1)
fl $\cdots$ Approximate value	(Real; Size: 1)
s $\cdots$ Work	(Real; Size: 1)
w $\cdots$ Work	(Real; Size: m+1)
id $\cdots$ Work index of a	(Integer; Size: 1)

the main portion of the program is as follows (double precision example):

```
fl = a[0];
if(m == 0){return 0;}
s = 1.0;
```

```
w[0] = 1.0;
for ( i = 1; i < (m + 1); i + +)
{
        s * =  xl;
        w[i]  =  s;
        fl + =  s * a[i];
}
for ( j = 1; j < (m + 1); j + +)
{
        for ( i = 0; i < (m + 2 − j); i + +)
        {
                w[i] * =  y1;
                id = (i + 1) + j * ×(2 × (m + 1) − j + 1)/2;
                fl + =  w[i] * a[id − 1];
        }
}
```

(7) **Example**

(a) Problem

Given the following input data:

k	x_k	y_k	z_k
1	6.95	−0.48	48.24
2	2.44	9.70	−17.57
3	0.89	−0.70	0.67
4	7.27	−7.51	38.75
5	−7.36	−1.18	53.82
6	−0.07	−4.72	−5.56
7	4.55	−5.84	12.18
8	−1.26	7.45	−12.29

obtain the approximation polynomial coefficients and approximate values at the input points.

(b) Input data

x, y, z, n = 8 and m = 2.

(c) Main program

```
/*      C interface example for ASL_dnrapl */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    double *z;
    int n;
    int m;
    double *a;
    double *f;
    int *iw;
    double *wk;
    int ierr;
    int i,niwk,nwk;
    FILE *fp;

    fp = fopen( "dnrapl.dat", "r" );
```

```
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dnrapl ***\n" );
printf( "\n     ** Input **\n\n" );
n=8;
x  = ( double * )malloc((size_t)( sizeof(double) * n ));
if( x  == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}
y  = ( double * )malloc((size_t)( sizeof(double) * n ));
if( y  == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}
z  = ( double * )malloc((size_t)( sizeof(double) * n ));
if( z  == NULL )
{
    printf( "no enough memory for array z\n" );
    return -1;
}
f  = ( double * )malloc((size_t)( sizeof(double) * n ));
if( f  == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}

for( i=0 ; i<n ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<n ; i++ )
{
    fscanf( fp, "%lf", &y[i] );
}
for( i=0 ; i<n ; i++ )
{
    fscanf( fp, "%lf", &z[i] );
}
fscanf( fp, "%d", &m );
niwk=((m+1)*(m+2))/2;
a  = ( double * )malloc((size_t)( sizeof(double) * niwk ));
if( a  == NULL )
{
    printf( "no enough memory for array a\n" );
    return -1;
}
iw  = ( int * )malloc((size_t)( sizeof(int) * niwk ));
if( iw  == NULL )
{
    printf( "no enough memory for array iw\n" );
    return -1;
}
nwk=((m+2)*(m+2)*(m+2)*(m+2))/4;
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tn  =%6d\n", n );
printf( "\tm  =%6d\n", m );
printf( "\n\t     x             y             z\n" );
for( i=0 ; i<n ; i++ )
{
    printf(  "\t%8.3g      %8.3g      %8.3g\n", x[i],y[i],z[i] );
}

fclose( fp );

ierr = ASL_dnrapl(x, y, z, n, m, a, f, iw, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n\tCoefficient of Polynomial for x,y\n\n" );
for( i=0 ; i<((m+1)*(m+2)/2) ; i++ )
{
    printf(  "\t  a[%6d] = %8.3g\n", i,a[i] );
}
printf( "\n\tApproximate Value of z\n\n" );
for( i=0 ; i<n ; i++ )
{
    printf(  "\t  f[%6d] = %8.3g\n", i,f[i] );
```

```
        }

        free( x );
        free( y );
        free( z );
        free( a );
        free( f );
        free( iw );
        free( wk );

        return 0;
    }
```

(d) Output results

```
*** ASL_dnrapl ***

 ** Input **

n  =     8
m  =     2

     x             y             z
    6.95         -0.48          48.2
    2.44           9.7         -17.6
    0.89          -0.7          0.67
    7.27         -7.51          38.8
   -7.36         -1.18          53.8
   -0.07         -4.72         -5.56
    4.55         -5.84          12.2
   -1.26          7.45         -12.3

 ** Output **

ierr =      0

Coefficient of Polynomial for x,y

  a[     0] =  0.00104
  a[     1] = -0.000183
  a[     2] =         1
  a[     3] = -0.000689
  a[     4] = 6.84e-05
  a[     5] =    -0.25

Approximate Value of z

  f[     0] =     48.2
  f[     1] =    -17.6
  f[     2] =    0.671
  f[     3] =     38.8
  f[     4] =     53.8
  f[     5] =    -5.56
  f[     6] =     12.2
  f[     7] =    -12.3
```

5.5.2 ASL_dngapl, ASL_rngapl
Two-Dimensional Lattice Data Least Squares Approximation Polynomial

(1) **Function**

If all Z coordinate values $z_{i,j}$ on two-dimensional lattice points (x_i, y_j) $(i = 1, \cdots, nx;\ j = 1, \cdots, ny)$ are given, ASL_dngapl or ASL_rngapl obtains the coefficients $a_{i,j}$ and the approximate values $f(x_i, y_j)$ $(i = 1, \cdots, nx;\ j = 1, \cdots, ny)$ at the input lattice points of the approximation polynomial for x and y:

$$f(x,y) = \sum_{i=1}^{ix+1} \sum_{j=1}^{iy+1} a_{i,j} x^{i-1} y^{j-1}$$

so that the sum of the squares of the differences of the polynomial values $f(x_i, y_j)$ and the Z coordinate values $z_{i,j}$ is minimized.

(2) **Usage**

Double precision:

```
ierr = ASL_dngapl  (x, nx, y, ny, z, ix, iy, a, f, wk);
```

Single precision:

```
ierr = ASL_rngapl  (x, nx, y, ny, z, ix, iy, a, f, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinates x_i of data points
2	nx	I	1	Input	Dimension nx of array x
3	y	{D∗ / R∗}	ny	Input	Y coordinates y_i of data points
4	ny	I	1	Input	Dimension ny of array y
5	z	{D∗ / R∗}	nx×ny	Input	Z coordinates $z_{i,j}$ at data points (x_i, y_j)
6	ix	I	1	Input	Highest degree ix for x of the approximation polynomial
7	iy	I	1	Input	Highest degree iy for y of the approximation polynomial
8	a	{D∗ / R∗}	See Contents	Output	Coefficients $a_{i,j}$ of approximation polynomial for x, y **Size**: $(\mathrm{ix}+1)\times(\mathrm{iy}+1)$
9	f	{D∗ / R∗}	nx×ny	Output	Approximate value $f(x_i, y_j)$ of Z coordinate at (x_i, y_j)
10	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: $9\times(\mathrm{nx}+2\times\mathrm{ny})$
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{nx} > 2, \mathrm{ny} > 2$

(b) $\mathrm{x[i]} \neq \mathrm{x[j]} \quad (\mathrm{i} \neq \mathrm{j})$
$\mathrm{y[i]} \neq \mathrm{y[j]} \quad (\mathrm{i} \neq \mathrm{j})$

(c) $0 < \mathrm{ix} \leq \min\,(8, \mathrm{nx}-1)$
$0 < \mathrm{iy} \leq \min\,(8, \mathrm{ny}-1)$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	

(6) **Notes**

(a) When Z coordinate values corresponding to two-dimensional lattice points (x_i, y_j) are $z_{i,j}$ $(i = 1, \cdots, \mathrm{nx};\ j = 1, \cdots, \mathrm{ny})$, $z_{i,j}$ are stored in array z as follows.
$\mathrm{z}[(\mathrm{i}-1)+\mathrm{nx}\times(\mathrm{j}-1)] = \mathrm{z_{i,j}}$

(b) The coefficients $a_{i,j}$ are stored in array a in the order
$a_{1,1}, \cdots, a_{1,iy+1}, a_{2,1}, \cdots, a_{2,iy+1}, \cdots, a_{ix+1,1}, \cdots, a_{ix+1,iy+1}$.

(c) Method of using the polynomial coefficients a to obtain an approximate value at a point other than the input data points.

xl	···X coordinate value of approximation point	(Real; Size: 1)	
yl	···Y coordinate value of approximation point	(Real; Size: 1)	
fl	···Approximate value	(Real; Size: 1)	
s	···Work	(Real; Size: 1)	the main portion of the
id	···Work index of a	(Integer; Size: 1)	
ie	···Work index of a	(Integer; Size: 1)	
k, lt	···Variables	(Integer; Size: 1)	

program is as follows:

```
id = (ix + 1) * (iy + 1);
fl = a[id - 1];
for ( lt = 1; lt < (iy + 1); i++)
{
        fl = fl * yl + a[id - l - 1];
}
for ( k = ix; k > 0; k--)
{
        ie = k * (iy + 1);
        s = a[ie - 1];
        for ( lt = 1; lt < iy + 1; lt--)
        {
                s = s * yl + a[ie - l - 1];
        }
        fl = fl * xl + s;
}
```

(d) Since this function approximates the given points on the surface by only one polynomial, meaningless results may be produced depending on the data. In particular, if the degree of the approximation polynomial is large, you should check whether or not the output results are appropriate. The fitting rate (See Section 5.5.1) is one characteristic for checking whether or not the output results are appropriate. If the fitting rate is low, use this function after subdividing the approximation interval and shifting or scaling the input data to the vicinity of the origin.

(7) **Example**

(a) Problem

Given the following lattice coordinates and corresponding z coordinate values as input data:

$x_1 = -3.0 \quad y_1 = -2.0$
$x_2 = -1.0 \quad y_2 = -1.0$
$x_3 = 1.0 \quad y_3 = 0.0$
$x_4 = 3.0 \quad y_4 = 1.0$
$y_5 = 2.0$

		j →				
	$z_{i,j}$	y_1	y_2	y_3	y_4	y_5
	x_1	28.0	29.5	27.0	20.5	10.0
$i\downarrow$	x_2	−2.0	2.5	3.0	−0.5	−8.0
	x_3	−8.0	−0.5	3.0	2.5	−2.0
	x_4	10.0	20.5	27.0	29.5	28.0

obtain the approximation polynomial coefficients and approximate values at the input lattice points.

(b) Input data

x, nx=4, y, ny=5, z, ix=2 and iy=2.

(c) Main program

```
/*      C interface example for ASL_dngapl */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    int nx;
    double *y;
    int ny;
    double *z;
    int ix;
    int iy;
    double *a;
    double *f;
    double *wk;
    int ierr;
    int i,j,nwk;
    FILE *fp;

    fp = fopen( "dngapl.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dngapl ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=4;
    ny=5;
    x  = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x  == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }
    y  = ( double * )malloc((size_t)( sizeof(double) * ny ));
    if( y  == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }
    z  = ( double * )malloc((size_t)( sizeof(double) * (nx*ny) ));
    if( z  == NULL )
    {
        printf( "no enough memory for array z\n" );
        return -1;
    }
    f  = ( double * )malloc((size_t)( sizeof(double) * (nx*ny) ));
    if( f  == NULL )
```

```
{
    printf( "no enough memory for array f\n" );
    return -1;
}

nwk=9*(nx+2*ny);
wk  = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk  == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<ny ; i++ )
{
    fscanf( fp, "%lf", &y[i] );
}
for( i=0 ; i<nx ; i++ )
{
    for( j=0 ; j<ny ; j++ )
    {
        fscanf( fp, "%lf", &z[i+nx*j] );
    }
}

fscanf( fp, "%d", &ix );
fscanf( fp, "%d", &iy );
a  = ( double * )malloc((size_t)( sizeof(double) * ((ix+1)*(iy+1)) ));
if( a  == NULL )
{
    printf( "no enough memory for array a\n" );
    return -1;
}
printf( "\tnx =%6d\n", nx );
printf( "\tny =%6d\n", ny );
printf( "\tix =%6d\n", ix );
printf( "\tiy =%6d\n", iy );
printf( "\n\tCoordinates \n" );
printf( "\t  x =" );
for( i=0 ; i<nx ; i++ )
{
    printf( "%8.3g ", x[i] );
}
printf( "\n\n" );
printf( "\t  y =" );
for( i=0 ; i<ny ; i++ )
{
    printf( "%8.3g ", y[i] );
}
printf( "\n\n" );
printf( "\t  z(x,y) = \n" );
for( i=0 ; i<nx ; i++ )
{
    printf( "\t     " );
    for( j=0 ; j<ny ; j++ )
    {
        printf( "%8.3g ", z[i+nx*j] );
    }
    printf( "\n" );
}

fclose( fp );

ierr = ASL_dngapl(x, nx, y, ny, z, ix, iy, a, f, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n\tCoefficient of Polynomial for x,y\n\n" );
for( i=0 ; i<((ix+1)*(iy+1)) ; i++ )
{
    printf(  "\t  a[%6d]=%8.3g\n", i,a[i] );
}
printf( "\n\tf(x,y) = \n" );
for( i=0 ; i<nx ; i++ )
{
    printf( "\t     " );
    for( j=0 ; j<ny ; j++ )
    {
        printf( "%8.3g ", f[i+nx*j] );
    }
    printf( "\n" );
}

free( x );
```

```
    free( y );
    free( z );
    free( a );
    free( f );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dngapl ***

 ** Input **

nx =     4
ny =     5
ix =     2
iy =     2

Coordinates
  x =      -3       -1        1        3

  y =      -2       -1        0        1        2

  z(x,y) =
            28     29.5       27     20.5       10
            -2      2.5        3     -0.5       -8
            -8     -0.5        3      2.5       -2
            10     20.5       27     29.5       28

 ** Output **

ierr =      0

Coefficient of Polynomial for x,y

  a[     0]=       0
  a[     1]=       0
  a[     2]=      -2
  a[     3]=       0
  a[     4]=     1.5
  a[     5]=       0
  a[     6]=       3
  a[     7]=       0
  a[     8]=       0

f(x,y) =
            28     29.5       27     20.5       10
            -2      2.5        3     -0.5       -8
            -8     -0.5        3      2.5       -2
            10     20.5       27     29.5       28
```

5.6 CHEBYSHEV APPROXIMATION

5.6.1 ASL_dncbpo, ASL_rncbpo Chebyshev Approximation

(1) **Function**

ASL_dncbpo or ASL_rncbpo obtains the Chebyshev coefficients c_j, which satisfy $F(x) \sim [\sum_{j=0}^{\mathrm{n}} \mathrm{c}_j \mathrm{t}_j(x)] - \frac{1}{2}\mathrm{c}_0$, by computing the Chebyshev approximation of a function $f(x)$ defined on the finite interval $[a, b]$ (when isw=0). Also, this function obtains the polynomial coefficients y_k for which $f(x) \sim \sum_{k=0}^{m} y_k x^k$ (when ISW=1).

The coefficients c_j are defined as $\mathrm{c}_j = \frac{\mathrm{n}}{2} \sum_{k=1}^{\mathrm{n}} [F[\cos(\pi(k - \frac{1}{2})/\mathrm{n}) \cos(\pi j(k - \frac{1}{2})/\mathrm{n})]$.

(2) **Usage**

Double precision:

```
ierr = ASL_dncbpo  (f, a, b, n, ceps, &aeps, c, &nc, isw, wk);
```

Single precision:

```
ierr = ASL_rncbpo  (f, a, b, n, ceps, &aeps, c, &nc, isw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	f	{D / R}	–	Input	Function f to be Chebyshev approximated (See Note (a))
2	a	{D / R}	1	Input	Lower end of approximation range a
3	b	{D / R}	1	Input	Upper end of approximation range b
4	n	I	1	Input	Required degree n of Chebyshev coefficients
5	ceps	{D / R}	1	Input	Chebyshev coefficient truncation error (See Note (c))
6	aeps	{D∗ / R∗}	1	Input	Admissible constant (See Note (b))
				Output	Estimating residual (See Note (b))
7	c	{D∗ / R∗}	n+1	Output	Chebyshev coefficients $C_j(j = 0, 1, \cdots, n)$ or polynomial coefficients $y_k(k$=0, 1, $\cdots, m)$
8	nc	I*	1	Output	Truncation degree m (See Note (c))
9	isw	I	1	Input	isw=0: Obtain only Chebyshev coefficients isw=1: Obtain Chebyshev coefficients and compute polynomial approximation according to those coefficients (See Note (c))
10	wk	{D∗ / R∗}	2∗(n+1)	Work	Work area
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) a < b

(b) n ≥ 0

(c) isw = 0 or isw=1

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Restriction (a) was not satisfied.	When a=b: c [0] =f(a) and c [i] =0.0 (i = 1, ⋯, n) are returned. When a > b: a and b are switched and processing continues.
3000	Restriction (b) was not satisfied.	Processing is aborted.
3010	Restriction (c) was not satisfied.	
3500	The error between the function and approximation polynomial is greater than the admissible constant aeps.	The error is output and the result that was obtained is returned.

(6) **Notes**

(a) The function f is created as follows.

```
double FORTRAN f(double *x)
{
    return (f(*x));
}
```

(b) The admissible constant is the maximum error predicted between the function $f(x)$ and the Chebyshev approximated function. The estimating residual is the maximum error estimated between the approximation function and original function $f(x)(\mathrm{a} \leq x \leq \mathrm{b})$ when the interval [a, b] is subdivided.

(c) The term "truncate" is used here with the following meaning. In order to compute a Chebyshev approximation optimally, the Chebyshev coefficients that were obtained are evaluated to obtain the degree nc(= m), which is the limit of the coefficients that cannot be ignored, and coefficients for degrees nc+1 and above are discarded.
The truncation error is the systematic error that always occurs when the formula containing the series is truncated at a specific term or according to a predetermined method.
The series is truncated only when the residual to be estimated is restrained by a certain predetermined admissible constant, and the more random error that occurs at that time is considered as part of the rounding error.

(7) **Example**

(a) Problem
Given the function $f(x) = x^2(x^2 - 2)\sin(x)$ on the interval $[-1, 1]$, obtain the coefficients of the Chebyshev approximation polynomial.

(b) Input data
Function: dnchev
Lower end of approximation range: a= − 1.0
Upper end of approximation range: b= 1.0
Required degree of Chebyshev coefficients: n=10

Chebyshev coefficient truncation error: ceps=0.01

Admissible constant: aeps=0.0001

Whether or not to compute polynomial approximation according to Chebyshev coefficients: isw=1

(c) Main program

```
/*      C interface example for ASL_dncbpo */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <asl.h>
#ifdef __cplusplus
extern "C"
{
#endif
#ifdef __STDC__
double f(double *x)
#else
double f(x)
double *x;
#endif
{
    return ((*x)*(*x)*((*x)*(*x)-2)*sin(*x));
}
#ifdef __cplusplus
}
#endif

int main()
{
    double a;
    double b;
    int n;
    double ceps;
    double aeps;
    double *c;
    int nc;
    int isw;
    double *wk;
    int ierr;
    int j;
    FILE *fp;

    fp = fopen( "dncbpo.dat", "r" );
    if( fp == NULL )
    {
           printf( "file open error\n" );
           return -1;
    }

    printf( "    *** ASL_dncbpo ***\n" );
    printf( "\n     ** Input **\n" );
    n = 10;
    isw = 1;

    fscanf( fp, "%lf", &a );
    fscanf( fp, "%lf", &b );
    fscanf( fp, "%lf", &ceps );
    fscanf( fp, "%lf", &aeps );

    c = ( double * )malloc((size_t)( sizeof(double) * (n+1) ));
    if( c == NULL )
    {
        printf( "no enough memory for array c\n" );
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * ((n+1)*2) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    printf( "\ta    = %8.3g \n", a);
    printf( "\tb    = %8.3g \n", b );
    printf( "\tn    =   %6d \n", n );
    printf( "\tceps = %8.3g \n", ceps );
    printf( "\taeps = %8.3g \n", aeps );
    printf( "\tisw  =   %6d \n", isw );

    fclose( fp );

    ierr = ASL_dncbpo(f, a, b, n, ceps, &aeps, c, &nc, isw, wk);

    printf( "\n     ** Output **\n" );
    printf( "\n\tierr =   %6d\n", ierr );
    printf( "\n\tnc   =   %6d\n", nc );
    printf( "\n\taeps = %8.3g \n", aeps );
```

```
    printf( "\n     ** POLYNOMIAL COEFFICIENT **\n" );
      for( j=0 ; j<nc+1 ; j ++ )
      {
      printf( "\tc[%6d]=          %8.3g\n", j,c[j] );
      }

    free( c );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dncbpo ***

 ** Input **
a    =       -1
b    =        1
n    =       10
ceps =   0.0001
aeps =     0.01
isw  =        1

 ** Output **

ierr =        0

nc   =        7

aeps = 3.19e-05

 ** POLYNOMIAL COEFFICIENT **
c[     0]=          -6.59e-17
c[     1]=          -0.00029
c[     2]=          6.45e-16
c[     3]=                -2
c[     4]=          -1.78e-15
c[     5]=              1.32
c[     6]=          1.33e-15
c[     7]=            -0.164
```

Chapter 6

SPLINE FUNCTIONS

6.1 INTRODUCTION

This chapter describes functions that perform interpolations or approximations of given data points using spline functions and that compute the derivatives and integrals of obtained spline function.
This library provides functions which perform interpolations or approximations of given data points using cubic spline functions (one-variable function), bicubic spline functions (two-variable function) and B–spline functions. In addition, an error detection and correction function is provided for when your input data is contaminated with isolated incongruous data values.
Normally, functions for a spline function are designed to obtain interpolation values, integrals or derivatives of spline function after obtaining spline coefficients. When obtaining a spline function, the following three category is set for distinguishing interpolation or approximation level.

(1) Interpolation
Create a spline function that passes through all given data points. In this case, all knots of a spline function is equal to given data points. This library provides several functions corresponding to methods for endpoints conditions decision.

(2) Smoothed interpolation
Create a spline function whose all knots abscissa values is equal to that of given data points. Obtained spline function may not passes through all given data points. It is not suitable for interpolation function for given data points, but smoother approximation function is obtained than one for simple interpolation. This method is effective when you want to obtain an approximation by revising all data points using the same compensation rate. Two methods are available. You can either enter a control variable corresponding to the compensation rate or automatically obtain the optimum compensation curve by a statistical procedure.

(3) Least squares interpolation
Create a spline function whose all knot positions are adjustable parameter. In this case, spline coefficients are obtained by applying least squares approximation in each interval between knots. More suitable spline function is obtained for some purpose than one for smoothed interpolation. Especially, when there are an extremely large number of data points, the least squares approximate spline function is effective. There are two methods of determining knot positions. You can specify them as input parameters or have them found automatically so that the total least squares error is minimized.

6.1.1 Notes

(1) There are innumerable functions which approximate (or interpolate) given data points. Usually obtained values from interpolations or approximations differ what kind of approximation function is chosen. What kind of approximation function is prefer depend on your purpose. A spline function is especially effective approximation function when you approximate or interpolate data points by smooth approximation function. There are several kind of spline functions and you must use proper one.

(2) The functions having names that begin ASL_[L]d/r store the spline coefficients (1st through 3rd order terms) in argument c as shown below:

Storage status within array c[3×(nx−1)]

$\hat{C}_{1,1}$	c[0]	$\hat{C}_{1,2}$	c[3]		$\hat{C}_{1,\mathrm{nx}-1}$	c[(nx−2)×3]
$\hat{C}_{2,1}$	c[1]	$\hat{C}_{2,2}$	c[4]	⋯	$\hat{C}_{2,\mathrm{nx}-1}$	c[(nx−2)×3+1]
$\hat{C}_{3,1}$	c[2]	$\hat{C}_{3,2}$	c[5]		$\hat{C}_{3,\mathrm{nx}-1}$	c[(nx−2)×3+2]
← − − − − − − − nx−1 − − − − − − − →						

3

Remarks

a. $\hat{C}_{1,j}$, $\hat{C}_{2,j}$ and $\hat{C}_{3,j}$ $(j = 1, 2, \cdots, \mathrm{nx}-1)$: the spline coefficients

However, for the functions having names that begin ASL_[L]w/v the spline coefficients are stored as follows:

Storage status within array c[3×nx]

$C_{1,1}$	c[0]	$C_{1,2}$	c[nx]	$C_{1,3}$	c[2×nx]
$C_{2,1}$	c[1]	$C_{2,2}$	c[nx+1]	$C_{2,3}$	c[2×nx+1]
$C_{3,1}$	c[2]	$C_{3,2}$	c[nx+2]	$C_{3,3}$	c[2×nx+2]
⋮	⋮	⋮	⋮	⋮	⋮
$C_{\mathrm{nx}-1,1}$		$C_{\mathrm{nx}-1,2}$		$C_{\mathrm{nx}-1,3}$	
0.0		0.0		0.0	
← − − − − − − − 3 − − − − − − − →					

nx-1

Remarks

a. $C_{i,1}$, $C_{i,2}$ and $C_{i,3}$ $(i = 1, 2, \cdots, \mathrm{nx}-1)$: the spline coefficients; $C_{i,k} = \hat{C}_{k,i}$ $(k = 1, 2, 3)$

Zeros are stored in cnx − 1, c2 × nx − 1, and c3 × nx − 1 and the actual spline coefficients are stored in c[i − 1 + nx × (j − 1)] i = 1, ⋯, nx − 1; j = 1, ⋯, 3. So, the size of the array c must be (nx×3). This change is for solving the simultaneous linear equation to obtain the spline coefficients effectively on vector computers.

(3) Since input data may contain error data, you should call the function that detects and corrects error data.

(4) Cubic spline coefficients for one-dimensional data differ according to the endpoint condition or smoothing method used. Therefore, to perform calculations using different conditions or smoothing methods, you must call the function that matches those conditions.

(5) To obtain values such as interpolation values, derivative values by cubic spline interpolation using the same endpoint conditions or smoothing methods repeatedly, processing will be more efficient if you call the function that obtains cubic spline coefficients once and then repeatedly call the function that obtains corresponding values by cubic spline coefficients.

(6) The bicubic spline function 6.3.1 {ASL_dgisxb / ASL_rgisxb} and 6.3.3 {ASL_dgiizb / ASL_rgiizb} do not obtain bicubic spline coefficients.

6.1.2 Algorithms Used

6.1.2.1 Cubic aperiodic spline function (inputting endpoint conditions)

A spline function is an interval-wise polynomial that connect m-th and lower order polynomials defined on each subinterval under the condition that the function values and $(m-1)$-th and lower derivatives are continuous on the entire interval. In general, the quadratic differential coefficients M_i are obtained as solutions of the following simultaneous linear equations.

$$\begin{bmatrix} 2 & \lambda_1 & & & & & \\ \mu_2 & 2 & \lambda_2 & & & 0 & \\ & \mu_3 & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \lambda_{n-2} & \\ & 0 & & & \mu_{n-1} & 2 & \lambda_{n-1} \\ & & & & & \mu_n & 2 \end{bmatrix} \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ \vdots \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{6.1}$$

If we assume that the X coordinates of the knots in each interval are given by:

$$-\infty < \xi_1 < \xi_2 < \cdots < \xi_n < \infty$$

the function values at each knot are:

$$y_1, \cdots, y_n$$

and the distance between knots is:

$$h_i = \xi_{i+1} - \xi_i$$

then:

$$\begin{aligned} \lambda_i &= \frac{h_i}{h_{i-1} + h_i} && \text{for } (i = 2, \cdots, n-1) \\ \mu_i &= 1 - \lambda_i && \text{for } (i = 2, \cdots, n-1) \\ d_i &= \frac{6}{h_{i-1} + h_i} \left\{ \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right\} && \text{for } (i = 2, \cdots, n-1) \end{aligned} \tag{6.2}$$

λ_1, μ_n, d_1 and d_n are determined from the endpoint conditions as described below.
If we let $f(x)$ represent the spline function, then:

(1) If $f'(\xi_1)$ and $f'(\xi_n)$ (slopes at the endpoints) are known:

$$\begin{aligned} &\lambda_1 = 1.0, \quad \mu_n = 1.0 \\ &d_1 = \frac{6}{h_1} \left\{ \frac{y_2 - y_1}{h_1} - f'(\xi_1) \right\} \\ &d_n = \frac{6}{h_{n-1}} \left\{ f'(\xi_n) - \frac{y_n - y_{n-1}}{h_{n-1}} \right\} \end{aligned} \tag{6.3}$$

(2) If $f''(\xi_1)$ and $f''(\xi_n)$ are know:

$$\begin{aligned} &\lambda_1 = \mu_n = 0.0 \\ &d_1 = 2f''(\xi_1) \\ &d_n = 2f''(\xi_n) \end{aligned} \tag{6.4}$$

If the endpoint conditions are not known, you can assume $f''(\xi_1) = f''(\xi_n) = 0.0$ and use the so-called cubic natural spline.

(3) If $f'''(\xi_1)$ and $f'''(\xi_n)$ are known:

$$\begin{aligned} &\lambda_1 = \mu_n = -2.0 \\ &d_1 = -2h_1 f'''(\xi_1) \\ &d_n = 2h_{n-1} f'''(\xi_n) \end{aligned} \tag{6.5}$$

It the endpoint conditions are not known, you can assume $f'''(\xi_1) = f'''(\xi_n) = 0.0$. This is called the P-spline.

You can solve these simultaneous linear equations as follows:

$b_i = 2 \quad \text{for} \quad (i = 1, \cdots, n)$

(Forward process)

$$\left.\begin{aligned} b_i &= b_i - \frac{\mu_i \lambda_{i-1}}{b_{i-1}} \\ d_i &= d_i - \frac{\mu_i d_{i-1}}{b_{i-1}} \end{aligned}\right\} \quad \text{for} \quad (i = 2, \cdots, n)$$

$M_n = d_n - b_n$

(Backward process)

$$M_i = \frac{d_i - \lambda_i M_{i+1}}{b_i} \quad \text{for} \quad (i = n-1, \cdots, 1)$$

Obtaining the spline coefficients $c_{1\sim3,i}$ from the derived M_i, we get:

$$\begin{aligned} c_{1,i} &= \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6}(M_{i+1} - M_i) - \frac{h_i M_i}{2} \\ c_{2,i} &= \frac{M_i}{2} \\ c_{3,i} &= \frac{M_{i+1} - M_i}{6h_i} \end{aligned} \tag{6.6}$$

6.1.2.2 Cubic periodic spline function

The simultaneous linear equations for obtaining the quadratic differential coefficients M_i of the periodic spline function are as follows.

$$\begin{bmatrix} 2 & \lambda_2 & \cdots & \cdots & 0 & \cdots & \mu_2 \\ \mu_3 & 2 & \lambda_3 & & & 0 & \vdots \\ \vdots & \mu_4 & \ddots & \ddots & & & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & \ddots & \ddots & \lambda_{n-2} & \vdots \\ \vdots & 0 & & & \mu_{n-1} & 2 & \lambda_{n-1} \\ \lambda_n & \cdots & 0 & \cdots & \cdots & \mu_n & 2 \end{bmatrix} \begin{bmatrix} M_2 \\ M_3 \\ \vdots \\ \vdots \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = \begin{bmatrix} d_2 \\ d_3 \\ \vdots \\ \vdots \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{6.7}$$

$M_1 = M_n, y_1 = y_n$

Use the following algorithm to solve these simultaneous linear equations.

$$
\begin{aligned}
b_1 &= 2 \\
b_i &= 2 - \lambda_i \frac{\mu_{i+1}}{b_{i-1}} && \text{for } (i = 2, \cdots, n-2) \\
p_i &= -\frac{\lambda_{i+1}}{b_i} && \text{for } (i = 1, \cdots, n-2) \\
q_1 &= -\frac{\mu_2}{2} \\
q_i &= -q_{i-1} \frac{\mu_{i+1}}{b_{i-1}} && \text{for } (i = 2, \cdots, n-2) \\
r_1 &= \frac{d_2}{2} \\
r_i &= \frac{d_{i+1} - r_{i-1}\mu_{i+1}}{b_{i-1}} && \text{for } (i = 2, \cdots, n-2) \\
t_i &= p_i t_{i+1} + q_i && (t_{n-1} = 1, \text{ for } (i = n-2, \cdots, 1)) \\
v_i &= p_i v_{i+1} + r_i && (v_{n-1} = 0, \text{ for } (i = n-2, \cdots, 1)) \\
M_n &= \frac{d_n - \lambda_n v_1 - \mu_n v_{n-2}}{\lambda_n t_1 + \mu_n t_{n-2} + 2} \\
M_i &= t_{i-1} M_n + v_{i-1} && \text{for } (i = n, \cdots, 2) \\
M_1 &= M_n \\
y_1 &= y_n
\end{aligned}
$$

The procedure for obtaining the spline coefficients from the derived M_i is similar to that described for aperiodic spline functions.

6.1.2.3 Cubic aperiodic spline functions (endpoint condition input is unnecessary)

In this case, use the algorithm that uses the first differential coefficients $m_i = f'(\xi_i)$. The simultaneous linear equations for obtaining m_i are as follows.

$$
\begin{bmatrix}
2 & \mu_1 & & & & & \\
\lambda_2 & 2 & \mu_2 & & & 0 & \\
 & \lambda_3 & \ddots & \ddots & & & \\
 & & \ddots & \ddots & \ddots & & \\
 & & & \ddots & \ddots & \mu_{n-3} & \\
 & 0 & & & \lambda_{n-2} & 2 & \mu_{n-2} \\
 & & & & & \lambda_{n-1} & 2
\end{bmatrix}
\begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ \vdots \\ \vdots \\ m_{n-2} \\ m_{n-1} \end{bmatrix}
=
\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{bmatrix}
\tag{6.8}
$$

λ and μ are similar to the case when endpoint conditions are input.

$$
c_i = 3 \left\{ \lambda_i \frac{(y_i - y_{i-1})}{h_{i-1}} + \mu_i \frac{(y_{i+1} - y_i)}{h_i} \right\} \quad \text{for } (i = 2, \cdots, n-2) \tag{6.9}
$$

Assume "not-a-knot" endpoint conditions. That is, assume that the third derivative $f'''(x)$ is continuous at x_1 through x_3 and x_{n-2} through x_n and the first and second derivatives are continuous at x_3 and x_{n-2}. The equations at the endpoints then become:

$$
\begin{aligned}
\mu_1 &= \frac{2(h_1 + h_2)}{h_2} \\
\lambda_{n-1} &= \frac{2h_{n-1}}{h_{n-1} + h_{n-2}} \\
c_1 &= \frac{2[(h_1 + 2(h_1 + h_2))(y_2 - y_1)/h_1 + {h_1}^2 (y_3 - y_2)/{h_2}^2]}{h_1 + h_2} \\
c_{n-1} &= \frac{2[{h_{n-2}}^2 (y_n - y_{n-1})/h_{n-1} + h_{n-1}(3h_{n-2} + 2h_{n-1})(y_{n-1} - y_{n-2})/h_{n-2}]}{(h_{n-1} + h_{n-2})^2}
\end{aligned}
\tag{6.10}
$$

Obtaining the spline coefficients $c_{1\sim3,i}$ from the derived m_i, we get:

$$\begin{cases} c_{1,i} = m_i \\ c_{2,i} = \dfrac{y_{i+1}-y_i}{{h_i}^2} - \dfrac{m_i}{h_i} - c_{3,i}h_i \\ c_{3,i} = \dfrac{m_{i+1}+m_i}{{h_1}^2} - 2\dfrac{y_{i+1}-y_i}{{h_i}^3} \end{cases} \tag{6.11}$$

6.1.2.4 Cubic spline smoothing by specifying a control variable

(See Reference Bibliography (5))

Approximate the data set by a smooth curve on which the data points are knots and consider a natural spline ($f''(\xi_1) = f''(\xi_n) = 0$) that minimizes the following function:

$$\begin{aligned} S_i &= \int_{\xi_1}^{\xi_n} (f''(x))^2 dx \\ S_m &= \sum_{i=1}^{n} \left(\frac{f(\xi_1) - y_i}{\delta y_i} \right)^2 \\ & S_i + pS_m \to \min \end{aligned} \tag{6.12}$$

where let S_f, which is the value that enables S_m to be obtained instead of specifying p, be the value of the control variable that is input. By varying the value of p under the condition of equation (6.12), determine the value of p such that $S_m = S_f$ be fulfilled.

Let δy_i be the estimated value of the deviation from y_i and choose values of S_f in the range:

$$N - \sqrt{2N} \le S_f \le N + \sqrt{2N}$$

N is the number of the data.

Let the components $q_{i,j}$ of the tridiagonal matrix Q be:

$$q_{i-1,i} = \frac{1}{h_{i-1}}, \quad q_{i,i} = -\frac{1}{h_{i-1}} - \frac{1}{h_i}, \quad q_{i+1,i} = \frac{1}{h_i} \tag{6.13}$$

the components $t_{i,j}$ of the tridiagonal matrix T be:

$$t_{i,j} = \frac{2}{3}(h_{i-1} + h_i), \quad t_{i,i+1} = t_{i+1,i} = \frac{h_i}{3} \tag{6.14}$$

and the components of the diagonal matrix D be:

$$d_i = \delta y_i \tag{6.15}$$

Using these notations, the algorithm for obtaining the spline coefficients is as follows.

① Start from $\bar{p} = 0$.

② Obtain the tridiagonal matrix $Q^T DQ$, the matrix $Q^T \boldsymbol{y}$, and the tridiagonal matrix T. ($\boldsymbol{y}$ is a vector having y_i as its component)

③ Perform the Cholesky decomposition $\bar{p}Q^T D^2 Q + T = R^T R$ and obtain the solution $\boldsymbol{u}$ by solving the simultaneous linear equation $R^T R\boldsymbol{u} = Q^T \boldsymbol{y}$ using forward substitution and back substitution.

④ Let $\boldsymbol{v} = \dfrac{\boldsymbol{y} - \boldsymbol{s}}{\bar{p}} = D^2 Q\boldsymbol{u}$ and obtain $\boldsymbol{v}$. ($\boldsymbol{s}$ is a vector having spline function value $f(\xi_i)$ at each knot.)
Obtain:

$$e = \boldsymbol{v}Q\boldsymbol{u} = \frac{(\boldsymbol{y} - \boldsymbol{s})^2}{\bar{p}D^2}, \quad S'_m = \bar{p}^2 e = \sum \left(\frac{f(\xi_i) - y_i}{\delta y_i} \right)^2$$

⑤ If $S'_m > S_f$, obtain a new $\bar{p}$ by the processing described below and then return to ③. Use Cholesky forward substitution to obtain the solution $\boldsymbol{g}$ of $R^T\boldsymbol{g} = Q^T D^2 Q\boldsymbol{u}$.

$$\begin{aligned} f &= \boldsymbol{g}^T\boldsymbol{g} \\ h &= e - \bar{p}f \\ \bar{p} &= \bar{p} + \frac{S_f - S'_m}{(\sqrt{S_f/e} + \bar{p})h} \end{aligned}$$

⑥ If $S'_m \leq S_f$, then end by calculating the spline coefficients as described below.

$$\begin{aligned} \boldsymbol{s} &= \boldsymbol{y} - \bar{p}\boldsymbol{v} \\ c_{2,i} &= u_i \\ h_i &= x_{i+1} - x_i \\ c_{3,i} &= \frac{u_{i+1} - u_i}{3h_i} \\ c_{1,i} &= \frac{f_{i+1} - f_i}{h_i} - c_{2,i}h_i - c_{3,i}{h_1}^2 \end{aligned}$$

6.1.2.5 Cubic spline automatic smoothing

(See Reference Bibliography (4))
To obtain the best approximation curve for the data set, perform the smoothing using the algorithm described below to obtain the value p (See preceding section) that minimizes the cross-validation function.

① Obtain $T^{1/2}$ as follows. (See equation (6.14) for matrix T)

$$\begin{aligned} T^{1/2} &= UXU^T \quad \text{(Singular-value decomposition)} \\ &\quad \left(\begin{array}{l} U : \text{Matrix consisting of eigenvectors of } T. \\ X : \text{Diagonal matrix having square roots of the eigenvalues} \\ \quad (\lambda_i) \text{ of } T \text{ as diagonal elements.} \end{array} \right) \\ T^{-1/2} &= UEU^T \\ &\quad (E : \text{Diagonal matrix having } \frac{1}{\sqrt{\lambda_i}} \text{ as diagonal components.}) \end{aligned}$$

② Obtain F from $F = QT^{-1/2}$. (See equation (6.13) for matrix Q)

③ Obtain $F = UWV^T$ by performing a singular-value decomposition of F. If $\boldsymbol{s}$ (function value at knot) $= A\boldsymbol{y}$, then:

$$\begin{aligned} I - A &= Q(Q^TQ + pT)^{-1}Q^T \\ &= F(F^TF + pI)^{-1}F^T \quad \text{(Assume Weight} = 1.) \\ I - A &= U \begin{pmatrix} \frac{{d_1}^2}{{d_1}^2 + p} & & 0 \\ & \ddots & \\ 0 & & \frac{{d_{n-2}}^2}{{d_{n-2}}^2 + p} \end{pmatrix} U^T \quad (d_i : \text{Diagonal element of } W) \end{aligned}$$

④ Let the approximation function $V(\hat{p})$ of the cross validation function be:

$$V(\hat{p}) = \frac{1}{n}\sum_{j=1}^{n-2}\left(\frac{{d_j}^2}{{d_j}^2+p}\right){z_j}^2 \Big/ \left[\frac{1}{n}\sum_{j=1}^{n-2}\left(\frac{{d_j}^2}{{d_j}^2+p}\right)\right]^2$$

where $\boldsymbol{z} = U^T\boldsymbol{y}$

Obtain the minimum value of $V(\hat{p})$ by using the Davidon's method to search for extremum values. Calculate the spline coefficients by obtaining the quadratic differential coefficients and the function values at knots from the value of $\hat{p}$ at the point where the minimum occurs. (See preceding section)
In the Davidon's method of searching for extremum values, first determine two points between which minimum point exists. Then interpolate the function by cubic polynomial using the function values and its derivatives at these points and obtain minimum value.

6.1.2.6 Cubic spline coefficients (least squares method with specification of knot locations)

(See Reference Bibliography (6))
Obtain the spline function by specifying interpolation intervals according to knots (ξ_i) so that the least squares error between the input data (y_i) and spline function values $f(x_i)$ is minimized.
Let $S = \sqrt{\sum_{i=1}^{n} w_i(f(x_i)-y_i)^2}$ be the least squares error, where the weight w_i at each data point has the value as follows.

$$\begin{aligned} w_1 &= \frac{x_2 - x_1}{x_n - x_1} \\ w_i &= \frac{x_{i+1} - x_{i-1}}{x_n - x_1} \quad \text{for} \quad (i = 2, \cdots, n-1) \\ w_n &= \frac{x_n - x_{n-1}}{x_n - x_1} \end{aligned} \tag{6.16}$$

Assume the spline function is represented by the concurrently orthonormal base functions $\{\Psi_i\}_{i=1}^m$.

$$\begin{aligned} &< \Psi_i, \Psi_j >= \delta_{ij}, \quad \text{for} \quad (i, j = 1, \cdots, m) \\ &(\delta \text{ indicates the Kronecker delta and } <> \text{ indicates the inner product.}) \end{aligned} \tag{6.17}$$

The function value u is $u = \sum_{i=1}^{m} < u, \Psi_i > \Psi_i$ where $< u, \Psi_i >$ is the coefficient of Ψ_i.
Consider the cubic spline base $\{\Phi_i\}_{i=1}^m$ and use the modified Gram-Schmidt orthogonalization method to obtain the orthonormal polynomial base from them. This method is expressed as follows.

$$\left.\begin{aligned} {\Phi_i}^{(1)} &= \Phi_i \\ {\Phi_i}^{(j+1)} &= {\Phi_i}^{(j)} - < {\Phi_i}^{(j)}, \Psi_j > \Psi_j \quad \text{for} \quad (j = 1, \cdots, i-1) \\ \Psi_i &= \frac{{\Phi_i}^{(i)}}{\|{\Phi_i}^{(i)}\|} \quad (\|{\Phi_i}^{(i)}\| = (< {\Phi_i}^{(i)}, {\Phi_i}^{(i)} >)^{1/2}) \end{aligned}\right\} \text{for} \quad (i = 1, \cdots, m) \tag{6.18}$$

If orthogonal polynomials are created as described above, then the approximation function for the added knots should be updated by adding the new orthogonal polynomial base and their coefficients. To obtain the function values and first differential coefficients at the added knots, values obtained from the new polynomial components should be added to the values before the knots were added.
This algorithm is described below divided into individual processing steps.

① Calculate weights.

② Use orthogonal polynomials to obtain the cubic least squares approximation polynomial on the single interval from the first knot ξ_0 to the last knot ξ_m. Let this bases be Ψ_1 through Ψ_4. In addition, obtain the least squares error for this.

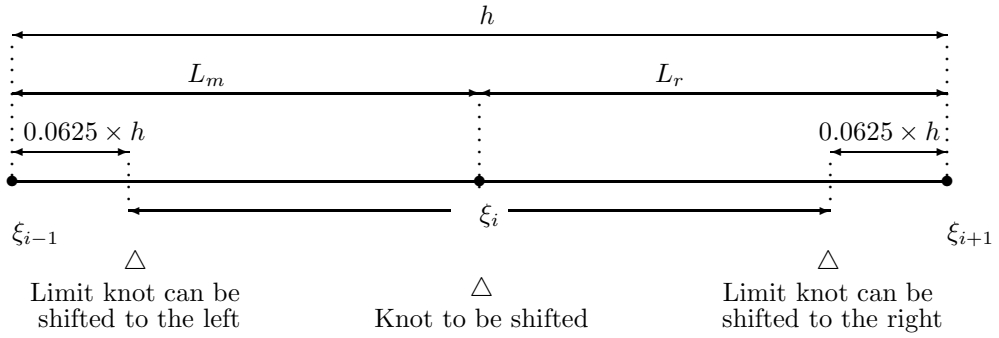
③ Next, add knots one at a time and calculate the cubic spline Φ_i; corresponding to the added knots, and obtain the orthonormal polynomial base by the modified Gram-Schmidt orthogonalization method. At the same time, calculate the function values and first differential coefficient values at the added knots and update the least squares error.

④ Finally, when all knots have been added, obtain the spline polynomial from the function values and first differential coefficient values.

⑤ When adding and modifying knots, create orthonormal polynomials from the polynomials and, at the end, regenerate the spline polynomial.

6.1.2.7 Cubic spline coefficients (least squares method with knot positions automatically determined)

(See Reference Bibliography (7))
The number of knots is fixed in advance and the least squares error E_S is minimized by moving interior knots ξ_i for $(i = 2, 3, \cdots, n-1)$.
The algorithm method is as follows. Fix the knots at the two endpoints and move knots sequentially from the rightmost knot to the leftmost knot to optimum positions so that the least squares error E_i on each interval is minimized.
The following figure shows the distance knots are shifted to the left or right.



Amount of shift to the left $=L_m\times$ (Average over the entire interval of ((Distance preceding knot was shifted)/h)). Initial value is $0.4\times L_m$

Amount of shift to the right $=L_r\times$ (Average over the entire interval of ((Distance preceding knot was shifted)/h)). Initial value is $0.4\times L_r$

6.1.2.8 Interpolation values according to cubic spline coefficients

Search the interval $\xi_i \leq x < \xi_{i+1}$ for the interpolation point. Then calculate the interpolation value by using the following interpolation equation.

$$f(x) = s_i + c_{1,i}(x - \xi_i) + c_{2,i}(x - \xi_i)^2 + c_{3,i}(x - \xi_i)^3 \tag{6.19}$$

s_i: The spline function value $f(\xi_i)$ at the knot. This equals y_i if neither smoothing nor the least squares methods is used.

6.1.2.9 Derivatives according to cubic spline coefficients

In the same way as getting interpolation values, search the interval for locations to differentiate and use the following equations.

$$\begin{aligned} f'(x) &= c_{1,i} + 2c_{2,i}(x-\xi_i) + 3c_{3,i}(x-\xi_i)^2 \\ f''(x) &= 2c_{2,i} + 6c_{3,i}(x-\xi_i) \end{aligned} \tag{6.20}$$

6.1.2.10 Integrals according to cubic spline coefficients

Let $[a, b]$ be the integration interval where a and b are in the following intervals.

$$\xi_i \le a < \xi_{i+1}, \xi_j \le b < \xi_{j+1}$$

Then:

$$\begin{aligned} \int_a^b f(x)dx &= \int_{\xi_i}^{\xi_{i+1}} f(x)dx - \int_{\xi_i}^{a} f(x)dx + \int_{\xi_{i+1}}^{\xi_j} f(x)dx + \int_{\xi_j}^{b} f(x)dx \\ &= \sum_{k=i}^{j-1} \int_{\xi_k}^{\xi_{k+1}} f(x)dx - \int_{\xi_i}^{a} f(x)dx + \int_{\xi_j}^{b} f(x)dx \end{aligned} \tag{6.21}$$

where:

$$\begin{aligned} \int_{\xi_k}^{\xi_{k+1}} f(x)dx &= s_i h_i + \frac{c_{1,i}}{2}{h_i}^2 + \frac{c_{2,i}}{3}{h_i}^3 + \frac{c_{3,i}}{4}{h_i}^4 \\ &= \frac{h_i}{2}(y_{i+1} + y_i) - \frac{{h_i}^3}{12}(c_{2,i+1} + c_{2,i}) \\ \int_{\xi_k}^{a} f(x)dx &= s_i(a-\xi_k) + \frac{c_{1,i}}{2}(a-\xi_k)^2 + \frac{c_{2,i}}{3}(a-\xi_k)^3 + \frac{c_{3,i}}{4}(a-\xi_k)^4 \end{aligned}$$

6.1.2.11 Bicubic spline coefficients

A bicubic spline is defined as an extension of a cubic spline. First, create a lattice partitioning of the X, Y plane according to the following expressions.

$$\begin{aligned} a < x_1 < x_2 < \cdots < x_m = b \\ c < y_1 < y_2 < \cdots < y_m = d \end{aligned} \tag{6.22}$$

Let $z_{i,j}$ be the function value at x_i, y_i.

(1) First, fix $y = y_k$ and interpolate by a cubic spline using data points $(x_i, z_{i,k})$ for $(i = 1, 2, \cdots, m)$ and "not-a-knot" endpoint conditions. Use the obtained cubic coefficient to obtain the derivative value $\dfrac{\partial z_{i,k}}{\partial x}$ for $(i = 1, \cdots, m)$ at the point (x_i, y_k).
Do this for $k = 1, \cdots, n$.

(2) Similarly, fix $x = x_k$, interpolate by a cubic spline using data points $(y_j, z_{k,j})$ for $(j = 1, \cdots, n)$ and "not-a-knot" endpoint conditions, and obtain the derivative values

$$\frac{\partial z_{k,i}}{\partial y} \text{ for } (i = 1, \cdots, n; k = 1, \cdots, m)$$

(3) Assume "not-a-knot" endpoint conditions at the four corners and interpolate by a cubic spline using the derivatives $\frac{\partial z_{i,j}}{\partial x}, \frac{\partial z_{i,j}}{\partial y}$ for $(i = 1, \cdots, m; j = 1, \cdots, n)$ obtained in steps 1 and 2 as data values. Use the obtained cubic spline to obtain the derivatives $\frac{\partial^2 z_{i,j}}{\partial x \partial y}$ for $(i = 1, \cdots, m; j = 1, \cdots, n)$ at the points (x_i, y_j).

You can simplify steps (1) through (3) described above by calculating cubic spline interpolation coefficients using first differential coefficients as follows.

(1) Obtain solutions $z_x(i, j)$ of the simultaneous linear equations:

$$h_i z_x(i+1, j) + 2(h_i + h_{i+1}) z_x(i, j) + h_{i+1} z_x(i-1, j) = 3 \left\{ \frac{h_i}{h_{i+1}} (z_{i+1,j} - z_{i,j}) + \frac{h_{i+1}}{h_i} (z_{i,j} - z_{i-1,j}) \right\} \quad (i = 2, \cdots, m)$$

for $j = 1, \cdots, n$, and:

$$h_i z_{xy}(i+1, j) + 2(h_i + h_{i+1}) z_{xy}(i, j) + h_{i+1} z_{xy}(i-1, j) = 3 \left\{ \frac{h_i}{h_{i+1}} (z_y(i+1, j) - z_y(i, j)) + \frac{h_{i+1}}{h_i} (z_y(i, j) - z_y(i-1, j)) \right\} \quad (i = 2, \cdots, m)$$

for $j = 1$ and $j = n$ (two endpoints).

(2) Obtain solutions $z_y(i, j)$ of:

$$k_j z_y(i, j+1) + 2(k_j + k_{j+1}) z_y(i, j) + k_{j+1} z_y(i, j+1) = 3 \left\{ \frac{k_j}{k_{j+1}} (z_{i,j+1} - z_{i,j}) + \frac{k_{j+1}}{k_j} (z_{i,j} - z_{i,j-1}) \right\} \quad (j = 1, \cdots, n)$$

for $i = 1, \cdots, m$.

(3) Obtain solutions $z_{xy}(i, j)$ of:

$$k_j z_{xy}(i, j+1) + 2(k_j + k_{j+1}) z_{xy}(i, j) + k_{j+1} z_{xy}(i, j+1) = 3 \left\{ \frac{k_j}{k_{j+1}} (z_x(i, j+1) - z_x(i, j)) + \frac{k_{j+1}}{k_j} (z_x(i, j) - z_x(i, j-1)) \right\} \quad (j = 1, \cdots, n)$$

for $i = 1, \cdots, m$.

6.1.2.12 Bicubic spline interpolation values

The function value at the point (x, y) is represented by the following equation.

$$f(x, y) = \sum_{\ell=0}^{3} \sum_{r=0}^{3} \alpha_{\ell,r}^{i,j} \left(\frac{x - x_i}{x_{i+1} - x_i} \right)^{\ell} \left(\frac{y - y_j}{y_{j+1} - y_j} \right)^{r} \tag{6.23}$$

$\alpha_{\ell,r}^{i,j}$ are obtained from the following matrix calculations:

$$\begin{aligned}
&\Gamma_{ij} = A(h_i) K_{ij} A(k_j)^T \\
&\Gamma_{ij} \quad = \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \quad \text{(Individual elements } \alpha_{\ell r}\text{)} \\
&A(t) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & t & 0 & 0 \\ -3 & -2t & 3 & -t \\ 2 & t & -2 & t \end{bmatrix} \\
&K_{ij} = \begin{bmatrix} z(i,j) & z_y(i,j) & z(i,j+1) & z_y(i,j+1) \\ z_x(i,j) & z_{xy}(i,j) & z_x(i,j+1) & z_{xy}(i,j+1) \\ z(i+1,j) & z_y(i+1,j) & z(i+1,j+1) & z_y(i+1,j+1) \\ z_x(i+1,j) & z_{xy}(i+1,j) & z_x(i+1,j+1) & z_{xy}(i+1,j+1) \end{bmatrix}
\end{aligned} \tag{6.24}$$

where:

$$\begin{aligned}
&x_i < x < x_{i+1} \quad , \quad y_j < y < y_{j+1} \\
&h_i = x_{i+1} - x_i \quad , \quad k_j = y_{j+1} - y_j \\
&z(i,j) = z_{i,j} \quad , \quad z_x(i,j) = \frac{\partial z_{i,j}}{\partial x} \quad , \quad z_y(i,j) = \frac{\partial z_{i,j}}{\partial y} \quad , \quad z_{xy}(i,j) = \frac{\partial^2 z_{i,j}}{\partial x \partial y}
\end{aligned}$$

In this library, the following values are set as spline coefficients.

$$\begin{aligned}
C(1,I,1,J) &= z_{i,j} &&= z(i,j) \\
C(2,I,1,J) &= \frac{\partial z_{i,j}}{\partial x} &&= z_x(i,j) \\
C(1,I,2,J) &= \frac{\partial z_{i,j}}{\partial y} &&= z_y(i,j) \\
C(2,I,2,J) &= \frac{\partial^2 z_{i,j}}{\partial x \partial y} &&= z_{xy}(i,j)
\end{aligned}$$

Therefore, $\alpha_{\ell,r}^{i,j}$ is not used to obtain spline function values. Instead, these values are obtained directly by combining the calculations in equations (6.23) and (6.24).

$$\begin{aligned}
m &\leftarrow 1 \\
h_x &\leftarrow x_{i+1} - x_i \\
h_y &\leftarrow y_{j+1} - y_j \\
U &\leftarrow (x - x_i)/h_x \\
V &\leftarrow (y - y_j)/h_y
\end{aligned}$$

for $k = j, j+1$
  for $\ell = 1, 2$
    $\alpha_0 \leftarrow C(1, i, \ell, k)$
    $\alpha_1 \leftarrow h_x C(2, i, \ell, k)$
    $\alpha_2 \leftarrow 3(C(1, i+1, \ell, k) - \alpha_0) - h_x C(2, i+1, \ell, k) - 2\alpha_1$
    $\alpha_3 \leftarrow 2(\alpha_0 - C(1, i+1, \ell, k)) + h_x C(2, i+1, \ell, k) + \alpha_1$
    $S_m \leftarrow \alpha_0 + U(\alpha_1 + U(\alpha_2 + U\alpha_3))$
    $m \leftarrow m + 1$

$$
\begin{aligned}
\alpha_0 &\leftarrow S_1 \\
\alpha_1 &\leftarrow h_y S_2 \\
\alpha_2 &\leftarrow 3(S_3 - \alpha_0) - h_y S_4 - 2\alpha_1 \\
\alpha_3 &\leftarrow 2(\alpha_0 - S_3) + h_y S_4 + \alpha_1 \\
f(x, y) &\leftarrow \alpha_0 + V(\alpha_1 + V(\alpha_2 + V\alpha_3))
\end{aligned}
$$

6.1.2.13 Bicubic spline mixed partial derivatives

$$
\begin{aligned}
f(x, y) &= \sum_{\ell=0}^{3}\sum_{r=0}^{3} \alpha_{\ell,r}^{i,j} \left(\frac{x - x_i}{x_{i+1} - x_i}\right)^{\ell} \left(\frac{y - y_j}{y_{j+1} - y_j}\right)^{r} \\
\frac{\partial f}{\partial x} &= \sum_{\ell=1}^{3}\sum_{r=0}^{3} -\ell\alpha_{\ell,r}^{i,j} \frac{(x - x_i)^{\ell-1}}{(x_{i+1} - x_i)^{\ell}} \left(\frac{y - y_j}{y_{j+1} - y_j}\right)^{r} \qquad (6.25) \\
\frac{\partial f}{\partial y} &= \sum_{\ell=0}^{3}\sum_{r=1}^{3} -r\alpha_{\ell,r}^{i,j} \left(\frac{x - x_i}{x_{i+1} - x_i}\right)^{\ell} \frac{(y - y_j)^{r-1}}{(y_{j+1} - y_j)^{r}} \qquad (6.26) \\
\frac{\partial^2 f}{\partial x \partial y} &= \sum_{\ell=1}^{3}\sum_{r=1}^{3} \ell r\alpha_{\ell,r}^{i,j} \frac{(x - x_i)^{\ell-1}}{(x_{i+1} - x_i)^{\ell}} \frac{(y - y_j)^{r-1}}{(y_{j+1} - y_j)^{r}} \qquad (6.27)
\end{aligned}
$$

These calculations are similar to those performed for interpolation values.

6.1.2.14 Bicubic spline double integral

Compute the double integral for the rectangular interval formed from integration intervals $[A, B]$ in the X direction and $[C, D]$ in the Y direction by summing integrals computed for subintervals or lattices contained in this interval. Use the following equation to compute the integral for a single interval.

$$
\begin{aligned}
&\int_{x_i}^{x}\int_{y_j}^{y} f(x, y)dydx \\
&= \sum_{\ell=0}^{3}\sum_{r=0}^{3} \alpha_{\ell,r}^{i,j} \frac{(x_{i+1} - x_i)(y_{j+1} - y_j)}{(\ell+1)(r+1)} \left(\frac{x - x_i}{x_{i+1} - x_i}\right)^{\ell+1} \left(\frac{y - y_j}{y_{j+1} - y_j}\right)^{r+1}
\end{aligned} \qquad (6.28)
$$

6.1.2.15 Plane data interpolation

For interpolation first, perform a spline interpolation assuming the distance between data points and X coordinate as abscissa value and ordinate value, respectively, then determine the interpolated X coordinate. Next, perform a spline interpolation assuming the abscissa value as the above and Y coordinate as ordinate value, then determine the interpolated y ordinate. Output the interpolation values x and y obtained in the above.

For smoothing, perform a spline interpolation using the cubic spline automatic smoothing function. When the string of input data points takes the shape of an open curve, obtain the spline coefficient with the method requiring no endpoint condition input. When the string of input data points takes the shape of a closed curve, obtain the spline coefficient with the method of periodic conditions.

6.1.2.16 Interpolation using a B–spline function (one–dimensional)

Assume that the data (x_i, y_i) $(i = 0, 1, \cdots, N)$ is given in the region $R : a = x_0 \leq x \leq x_N = b$. At this time, use the following spline function of order m (degree $m - 1$):

$$S(x_i) = y_i \quad (i = 0, 1, \cdots, N) \tag{6.29}$$

having the following (internal) knots that were determined in advance:

$$\xi_1 \leq \xi_2 \leq \cdots \leq \xi_n \tag{6.30}$$

to perform interpolation. Since interpolation can be uniquely performed at this time, the condition:

$$N + 1 = m + n \tag{6.31}$$

and the Shoenberg–Whitney conditions must hold.
Here, the Shoenberg–Whitney conditions are that the following inequalities are satisfied:

$$\left.\begin{array}{l} x_0 < \xi_1 < x_m, \\ x_1 < \xi_2 < x_{m+1}, \\ \cdots\cdots\cdots\cdots \\ x_{N-m} < \xi_n < x_N \end{array}\right\} \tag{6.32}$$

A B–spline of order m (degree $m - 1$) is used as the basis function of the spline function $S(x)$. To create the required basis function, we introduce the following $2m$ additional knots:

$$\left.\begin{array}{l} \xi_{1-m} = \xi_{2-m} = \cdots = \xi_0 = a, \\ \xi_{n+1} = \xi_{n+2} = \cdots = \xi_{n+m} = b \end{array}\right\} \tag{6.33}$$

This corresponds to entering m overlaid knots on the endpoints of the interval $[a, b]$.
Once this is done, $S(x)$ is expressed in $\xi_0 \leq x \leq \xi_{n+1}$ as follows:

$$S(x) = \sum_{i=1}^{n+m} c_i^* M_{mi}(x) = \sum_{i=1}^{n+m} c_i N_{mi}(x) \tag{6.34}$$

Here, $M_{mi}(x)$ is the B–spline of order m defined for the knots $\xi_{i-m}, \xi_{i-m+1}, \cdots, \xi_i$. Also, $N_{mi}(x)$, which is the normalized B–spline, is defined by:

$$N_{mi}(x) = (\xi_i - \xi_{i-m}) M_{mi}(x) \tag{6.35}$$

The value of the B–spline $M_{mi}(x)$ can be easily calculated by using the de Boor–Cox algorithm. The de Boor–Cox algorithm is a computational technique that uses the following asymptotic expressions:

$$M_{rj}(x) = \frac{(x - \xi_{j-r}) M_{r-1,j-1}(x) + (\xi_j - x) M_{r-1,j}(x)}{\xi_j - \xi_{j-r}} \quad (r = 2, 3, \cdots, m) \tag{6.36}$$

$$M_{1j} = \begin{cases} \dfrac{1}{\xi_j - \xi_{j-1}} & (\xi_{j-1} \leq x < \xi_j) \\ 0 & (\text{Otherwise}) \end{cases} \tag{6.37}$$

By substituting (6.34) in (6.29), we obtain the following simultaneous linear equations:

$$\sum_{i=1}^{n+m} c_i N_{mi}(x_j) = y_j \quad (j = 0, 1, \cdots, N) \tag{6.38}$$

Let these equations be expressed in matrix representation as:

$$A\boldsymbol{c} = \boldsymbol{y} \tag{6.39}$$

Equation (6.39) can be effectively solved by using Gaussian elimination with partial pivoting. When the solution is substituted in (6.34), the interpolation spline $S(x)$ is determined.

6.1.2.17 Interpolation using a B–spline function (multi–dimensional)

Lets extend the interpolation method for one–dimensional data using a B–spline, which was described in (17), to the case for two–dimensional data. Assume that the function $f_{ij} = f(x_i, y_j)$ is given at the knots $(x_i, y_j)(i = 0, 1, \cdots, I; j = 0, 1, 2, \cdots, J)$ in the rectangular region $R : a = x_0 \leq x \leq x_I = b; c = y_0 \leq y \leq y_J = d$. At this time, use the following spline function of two variables of order m (degree $m - 1$) and order n (degree $n - 1$):

$$S(x_i, y_j) = f_{ij} \quad (i = 0, 1, \cdots, I; j = 0, 1, \cdots, J) \tag{6.40}$$

having the following (internal) knots in the x direction:

$$\xi_1 \leq \xi_2 \leq \cdots \leq \xi_h \tag{6.41}$$

and the following (internal) knots in the y direction:

$$\zeta_1 \leq \zeta_2 \leq \cdots \leq \zeta_k \tag{6.42}$$

to perform interpolation. Assume that the following relationships hold for the x direction:

$$I + 1 = h + m, \tag{6.43}$$

$$\left.\begin{array}{l} x_0 < \xi_1 < x_m \\ x_1 < \xi_2 < x_{m+1} \\ \cdots\cdots\cdots\cdots\cdots \\ x_{I-m} < \xi_h < x_I \end{array}\right\} \tag{6.44}$$

and the following relationships hold for the y direction:

$$J + 1 = k + n, \tag{6.45}$$

$$\left.\begin{array}{l} y_0 < \zeta_1 < y_n \\ y_1 < \zeta_2 < y_{n+1} \\ \cdots\cdots\cdots\cdots\cdots \\ y_{J-n} < \zeta_k < y_J \end{array}\right\} \tag{6.46}$$

The spline function $S(x, y)$ can be formed by using a set of basis functions. These basis functions can be created by taking tensor products of one–dimensional basis functions. To create the required basis function, we introduce the following $2m$ additional knots in the x direction:

$$\left.\begin{array}{l} \xi_{1-m} = \cdots = \xi_0 = a, \\ \xi_{h+1} = \cdots = \xi_{h+m} = b \end{array}\right\} \tag{6.47}$$

and the following $2n$ additional knots in the y direction:

$$\left.\begin{aligned}\zeta_{1-n} = \cdots = \zeta_0 = c,\\ \zeta_{k+1} = \cdots = \zeta_{k+n} = d\end{aligned}\right\} \tag{6.48}$$

At this time, the product $M_{mi}(x)M_{nj}(y)$ $(i = 1, 2, \cdots, h+m; j = 1, 2, \cdots, k+n)$ of $M_{mi}(x), (i = 1, 2, \cdots, h+m)$, which is the B–spline of order m (degree $m-1$) for the knot $\boldsymbol{\xi} = (\xi_{1-m}, \xi_{2-m}, \cdots, \xi_{h+m})$, and $M_{nj}(y), (j = 1, 2, \cdots, k+n)$, which is the B–spline of order n (degree $n-1$) for the knot $\boldsymbol{\zeta} = (\zeta_{1-n}, \zeta_{2-n}, \cdots, \zeta_{k+n})$, becomes the basis function of the spline function $S(x, y)$. That is, $S(x, y)$ is expressed in $\xi_0 \le x \le \xi_{h+1}, \zeta_0 \le y \le \zeta_{k+1}$ as follows:

$$S(x,y) = \sum_{i=1}^{h+m}\sum_{j=1}^{k+n} c_{ij}^{*} M_{mi}(x)M_{nj}(y) = \sum_{i=1}^{h+m}\sum_{j=1}^{k+n} c_{ij} N_{mi}(x)N_{nj}(y) \tag{6.49}$$

Here, $N_{mi}(x)$ and $N_{nj}(y)$, which are normalized B–splines of order m (degree $m-1$) and order n (degree $n-1$) respectively, satisfy the following relationships:

$$\left.\begin{aligned}N_{mi}(x) = (\xi_i - \xi_{i-m})M_{mi}(x)\\ N_{nj}(y) = (\zeta_j - \zeta_{j-n})M_{nj}(y)\end{aligned}\right\} \tag{6.50}$$

The values of these B–splines can be easily calculated by using the de Boor–Cox algorithm described in (17).
If we substitute (6.49) in (6.40), we obtain:

$$\sum_{r=1}^{h+m}\sum_{s=1}^{k+n} c_{rs}N_{mr}(x_i)N_{ns}(y_j) = f_{ij} \quad (i = 0, 1, \cdots, I; j = 0, 1, \cdots, J) \tag{6.51}$$

Equation (6.51), which represents simultaneous linear equations for which c_{rs} is unknown, has a unique solution because (6.43) to (6.46)are assumed.
Equation (6.51) can be expressed in matrix representation as:

$$\boldsymbol{Ac} = \boldsymbol{f} \tag{6.52}$$

where, $\boldsymbol{c}$ and $\boldsymbol{f}$ can be written as follows:

$$\boldsymbol{c} = [c_{11}, c_{21}, \cdots, c_{h+m,1}, \cdots, c_{h+m,k+n}]^T \tag{6.53}$$

$$\boldsymbol{f} = [f_{00}, f_{10}, \cdots, f_{I0}, \cdots, f_{IJ}]^T \tag{6.54}$$

Equation (6.52) can be solved by using Gaussian elimination with partial pivoting. When the solution is substituted in (6.49), the interpolation spline $S(x, y)$is determined.
The interpolation method for two–dimensional data described above can be extended in a similar manner for interpolation of multidimensional data of three or more dimensions.

6.1.2.18 B–spline smoothing (one–dimensional data)

Assume that data is given in the interval $[a, b]$ and that F_k is defined as follows:

$$F_k = f(x_k) + e_k \quad (k = 1, 2, \cdots, N) \tag{6.55}$$

where, $f(x)$ is the underlying function of the data (unknown function) and the e_k are mutually independent errors that obey a normal distribution with mean 0 and variance σ^2.

Use the least squares method to apply equation (6.34) to the data (6.55). Assume that the knots are given. The sum of the squares of the residuals is as follows:

$$Q = \sum_{k=1}^{N} \{S(x_k) - F_k\}^2 \tag{6.56}$$

Partially differentiate (6.56) with respect to parameter $c_i (i = 1, 2, \cdots, n + m)$ and set it to 0 to obtain the normalized equation:

$$A\boldsymbol{c} = \boldsymbol{d} \tag{6.57}$$

Here $\boldsymbol{c}$ and $\boldsymbol{d}$ are as follows:

$$\boldsymbol{c} = (c_1, c_2, \cdots, c_{n+m})^T, \tag{6.58}$$

$$\begin{aligned} \boldsymbol{d} = \{ &\sum_{x_k \in [\xi_{1-m}, \xi_1]} N_{m1}(x_k) F_k, \sum_{x_k \in [\xi_{2-m}, \xi_2]} N_{m2}(x_k) F_k, \cdots, \\ &\sum_{x_k \in [\xi_n, \xi_{n+m}]} N_{m,n+m}(x_k) F_k \}^T \end{aligned} \tag{6.59}$$

T represents the transpose.
The element in row i and column j of A is expressed as follows:

$$a_{ij} = \sum N_{mi}(x_k) N_{mj}(x_k) \tag{6.60}$$

The values of the B–spline $N_{mi}(x)$ can be easily calculated as described in (17). Also, by taking into account that the coefficient matrix A is a band matrix, (6.57) can be effectively solved by using Cholesky's method. If its solution $\boldsymbol{c}$ is substituted in (6.34), the approximation function $S(x)$ is determined.
To obtain a good approximation, the number of knots and their positions must be determined appropriately. Here, we assume that the standard AIC for applications is used, and that the best has been selected from among several applications that are considered. (That is, we assume that a good number of knots and good positions are obtained.)
This AIC, which is also called Akaike's Information Criterion, is defined as follows:

$$AIC = (-2) \log_e(\text{maximum likelihood}) + 2(\text{number of parameters}) \tag{6.61}$$

Here, the number of parameters is the number that can freely change within the statistical model. The model that minimizes the AIC value is considered to be the best model. When the AIC is used, a subjective judgement is completely unnecessary, and the best model from among multiple models that are considered can be automatically determined.
Let's consider the following regression model:

$$F_k = S(x_k) + e_k \quad (k = 1, 2, \cdots, N) \tag{6.62}$$

The parameters of this model are the coefficients $c_i (i = 1, 2, \cdots, n + m)$ of $S(x)$ and the variance σ^2 of the error. Since the knots of $S(x)$ are determined before the application of the model, the number of parameters cannot be increased. Therefore, from (6.61), AIC is as follows:

$$AIC = N \log_e Q + 2(n + m) \tag{6.63}$$

The model that minimizes (6.63) is considered to be the best approximation function. Therefore, we can try varying the number of knots and their positions to search for the application that minimizes AIC as much as possible. If there is a number of knots and corresponding positions for which that application can be satisfied, then we can assume that the best approximation was obtained.

6.1.2.19 B–spline smoothing (multi–dimensional data)

Let's extend the one–dimensional smoothing using a B–spline, which was described in (19), to the case for two–dimensional data. Assume that data is given in the rectangular region $R = [a, b] \times [c, d]$ on the $x - y$ plane and that F_{ij} is defined as follows:

$$F_{ij} = f(x_i, y_j) + e_r \quad (i = 1, 2, \cdots, I; j = 1, 2, \cdots, J; r = 1, 2, \cdots, IJ) \tag{6.64}$$

where, $f(x, y)$ is the underlying function of the data (unknown function) and the e_r are mutually independent errors that obey a normal distribution with mean 0 and variance σ^2. The sample points (x_i, y_j) are given at grid points.
Use the least squares method to apply equation (6.49) to the data (6.64). Assume that the knots are given in each direction. The sum of the squares of the residuals is as follows:

$$Q = \sum_{r=1}^{N} \{S(x_r, y_r) - F_r\}^2 \tag{6.65}$$

Partially differentiate (6.65) with respect to parameter c_{ij} $(i = 1, 2, \cdots, h + m; j = 1, 2, \cdots, k + n)$ and set it to 0 to obtain the normalized equation:

$$A\boldsymbol{c} = \boldsymbol{d} \tag{6.66}$$

Here $\boldsymbol{c}$ and $\boldsymbol{d}$ are as follows:

$$\boldsymbol{c} = (c_{11}, c_{21}, \cdots, c_{h+m,1}, \cdots, c_{h+m,k+n})^T \tag{6.67}$$

$$\begin{aligned} \boldsymbol{d} = \{ & \sum_{x_r \in [\xi_{1-m}, \xi_1] \cap y_r \in [\zeta_{1-n}, \zeta_1]} N_{m1}(x_r) N_{n1}(y_r) F_r, \\ & \sum_{x_r \in [\xi_{2-m}, \xi_2] \cap y_r \in [\zeta_{1-n}, \zeta_1]} N_{m2}(x_r) N_{n1}(y_r) F_r, \\ & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots, \\ & \sum_{x_r \in [\xi_h, \xi_{h+m}] \cap y_r \in [\zeta_{1-n}, \zeta_1]} N_{m,h+m}(x_r) N_{n1}(y_r) F_r, \\ & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots, \\ & \sum_{x_r \in [\xi_h, \xi_{h+m}] \cap y_r \in [\zeta_k, \zeta_{k+n}]} N_{m,h+m}(x_r) N_{n,k+n}(y_r) F_r \}^T \end{aligned} \tag{6.68}$$

By taking into account that the coefficient matrix A is a band matrix, (6.66) can be effectively solved by using Cholesky's method.
Let's consider the following regression model:

$$F_{ij} = f(x_i, y_j) + e_r \quad (i = 1, 2, \cdots, I; j = 1, 2, \cdots, J; r = 1, 2, \cdots, IJ) \tag{6.69}$$

From (6.61), AIC is as follows:

$$AIC = N \log_e Q + 2(h + m)(k + n) \tag{6.70}$$

We can try varying the number of knots and their positions to search for the application that minimizes AIC as much as possible. If there is a number of knots and corresponding positions for which that application can be satisfied, then we can assume that the best approximation was obtained. The smoothing for two–dimensional data described above can be extended in a similar manner for smoothing of multidimensional data of three or more dimensions.

6.1.3 Reference Bibliography

(1) De Boor, C. , "A Practical Guide to Splines", Springer-Verlag New York, (1978).

(2) Ahlberg, J. , Nilson, E. and Walsh, J. , "The Theory of Splines and Their Applications", Academic Press New York, (1967).

(3) Guerra, V. and Tapia, R. A. , "A Local Procedure for Error Detection and Data Smoothing", MRC Technical Summary Report #1452, Mathematics Research Center, University of Wisconsin-Madison, (1974).

(4) Craven, P. and Wahaba, G. , "Smoothing Noisy Data with Spline Functions", Numer. Math. , Vol. 31, pp. 377-403, (1979).

(5) Reinsch, C. H. , "Smoothing by Spline Functions", Numer. Math. , Vol. 10, pp. 177-183, (1967).

(6) De Boor, C. and Rice, J. R. , "Least Squares Cubic Spline Approximation I - Fixed Knots", Computer Sciences Department TR20, Purdue Univ. , (1968).

(7) De Boor, C. and Rice, J. R. , "Least Squares Cubic Spline Approximation II - Variable Knots", Computer Sciences Department TR21, Purdue Univ. , (1968).

(8) STONE, HAROLD. S. , "Parallel Tridiagonal Equation Solvers", ACM Transactions on Mathematical Software, Vol. 1, No. 4, 289 (1975).

(9) Hockney, R. W. and Jesshope, C. R. , "Parallel Computer"

6.2 CUBIC SPLINE (CURVED LINE INTERPOLATION)

6.2.1 ASL_dgispc, ASL_rgispc
Interpolation Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgispc or ASL_rgispc obtains cubic spline coefficients for "not-a-knot" endpoint conditions when endpoint conditions need not be entered and calculates interpolation values at specified points. The knots are set equal to sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgispc (x, y, n, xl, fl, m, c);

Single precision:

ierr = ASL_rgispc (x, y, n, xl, fl, m, c);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]) for $i = 1, \cdots,$ n (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
				Output	0th order terms of cubic spline coefficients (Same as input values)
3	n	I	1	Input	Number of sample points
4	xl	{D∗ / R∗}	m	Input	Abscissa values of points where interpolation values are calculated
5	fl	{D∗ / R∗}	m	Output	Cubic spline interpolation values at xl[i − 1]
6	m	I	1	Input	Number of interpolation values
7	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

(a) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\ \text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\ \text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\ \text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array x, y, c and variable n must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation $y_i = x_i e^{-4.0x_i}$ to find y_i $(i = 1, 2, \cdots, 9)$ at each point
$\{x_i\} = \{0.0, 0.1, 0.23, 0.34, 0.47, 0.59, 0.73, 0.92, 1.0\}$
and perform cubic spline interpolation using these points as sample points. In addition, obtain the value of the cubic spline function at the uniformly spaced points $xl_j = 0.1 \times j (j = 1, 2, \cdots, 10)$.

(b) Input data
x[i − 1]=x_i, y[i − 1]=y_i (i = 1, ··· , n), xl[j − 1]=xl_j (j = 1, ··· , m), n = 9 and m = 10.

(c) Main program

```
/*      C interface example for ASL_dgispc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int nx;
    double *u;
    double *s;
    int m;
    double *c;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dgispc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgispc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=9;
    m=10;

    c = ( double * )malloc((size_t)( sizeof(double) * (nx*m) ));
    if( c == NULL )
```

```
{
    printf( "no enough memory for array c\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}


y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

u = ( double * )malloc((size_t)( sizeof(double) * m ));
if( u == NULL )
{
    printf( "no enough memory for array u\n" );
    return -1;
}


s = ( double * )malloc((size_t)( sizeof(double) * m ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}


printf( "\tn = %6d\tm = %6d\n", nx, m );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &u[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &y[i] );
}

printf( "\n\tCoordinates (x,y)\n\n" );
printf( "\t     i     x[i]     y[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d %8.3g %8.3g\n", i,x[i],y[i] );
}

printf( "\n\tSpecified Points\n\n" );
printf( "\t     j    xl[j]\n");
for( j=0 ; j<m ; j++ )
{
    printf( "\t%6d %8.3g\n", j,u[j] );
}

fclose( fp );

ierr = ASL_dgispc(x, y, nx, u, s, m, c);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );

printf( "\n" );
for( j=0 ; j<m ; j++ )
{
    printf( "\tfl[%6d]=%8.3g\n", j,s[j] );
}

printf( "\n" );
printf( "\tc[i,j]\n" );
printf( "\t               i=0      i=1      i=2\n" );
for( j=0 ; j<nx-1 ; j++ )
{
    printf( "\t j=%6d",j );
    for( i=0 ; i<3 ; i++ )
    {
        printf( " %8.3g", c[i+3*j] );
    }
    printf( "\n" );
}
```

```
    free( x );
    free( y );
    free( u );
    free( s );
    free( c );

    return 0;
}
```

(d) Output results

```
*** ASL_dgispc ***

 ** Input **

n =      9  m =     10

Coordinates (x,y)

     i     x[i]     y[i]
     0        0        0
     1      0.1    0.067
     2     0.23   0.0917
     3     0.34   0.0873
     4     0.47   0.0717
     5     0.59   0.0557
     6     0.73   0.0394
     7     0.92   0.0232
     8        1   0.0183

Specified Points

     j    xl[j]
     0      0.1
     1      0.2
     2      0.3
     3      0.4
     4      0.5
     5      0.6
     6      0.7
     7      0.8
     8      0.9
     9        1

 ** Output **

ierr =      0

fl[     0]=   0.067
fl[     1]=  0.0901
fl[     2]=  0.0904
fl[     3]=  0.0808
fl[     4]=  0.0676
fl[     5]=  0.0544
fl[     6]=  0.0426
fl[     7]=  0.0326
fl[     8]=  0.0246
fl[     9]=  0.0183

c[i,j]
                 i=0      i=1      i=2
 j=     0    0.963    -3.29     3.66
 j=     1    0.414     -2.2     3.66
 j=     2   0.0281   -0.769     1.36
 j=     3  -0.0917    -0.32    0.783
 j=     4   -0.135  -0.0145    0.245
 j=     5   -0.128   0.0736   0.0659
 j=     6   -0.104    0.101  -0.0269
 j=     7   -0.068    0.086  -0.0269
```

6.2.2 ASL_dgissc, ASL_rgissc
Smoothed Interpolation Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgissc or ASL_rgissc automatically obtains the optimum smoothed cubic spline coefficients and calculates interpolation values at specified points. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgissc  (x, yd, n, xl, fl, m, y, c, isw, wk);
```

Single precision:

```
ierr = ASL_rgissc  (x, yd, n, xl, fl, m, y, c, isw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]) for $i = 1, \cdots,$ n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xl	{D∗ / R∗}	m	Input	Abscissa values of points where interpolation values are calculated
5	fl	{D∗ / R∗}	m	Output	Smoothed cubic spline interpolation values at xl[i − 1]
6	m	I	1	Input	Number of interpolation values
7	y	{D∗ / R∗}	n	Output	0th order terms of cubic spline coefficients
8	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
9	isw	I	1	Input	Processing switch isw=1: When the abscissa value spacing may not be uniform. When sample points are nearly uncorrelated, limit value n up to 20. isw=2: When the abscissa value spacing is uniform.
10	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: n × (2 × n + 4) (when isw=1) 6 × n (when isw=2)
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 4

(b) x[0] < x[1] < ⋯ < x[n − 1] (Ascending order)

(c) If isw=2, abscissa values must be uniformly spaced.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
4000	The minimum cross-validation value could not be found. (The data is uncorrelated.)	

(6) **Notes**

(a) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\\text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\\text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\\text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array x, y, c and variable n must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation $y_i = \sin(3\pi x_i/2) + e_i$ (e_i:Uniform random number in the interval $[-0.2, 0.2]$) to find values y_i at each point $x_i = (i-1)/24$ ($i = 1, 2, \cdots, 25$) and perform smoothed cubic spline interpolation using these points as sample points. In addition, obtain the value of the cubic spline function at $xl_j = 0.1 \times (j-1) (j = 1, 2, \cdots, 10)$.

(b) Input data
x[i − 1]=x_i, yd[i − 1]=y_i (i = $1, \cdots,$ n), xl[j − 1]=xl_j (j = $1, \cdots,$ m), n=25, isw=2 and m=10.

(c) Main program

```
/*      C interface example for ASL_dgissc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *f;
    int nx;
    double *u;
    double *s;
    int m;
    double *y;
    double *c;
    int isw;
    double *wk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dgissc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }
```

```
printf( "    *** ASL_dgissc ***\n" );
printf( "\n     ** Input **\n\n" );
nx=25;
m=10;
isw=2;

c = ( double * )malloc((size_t)( sizeof(double) * (3*(nx-1)) ));
if( c == NULL )
{
    printf( "no enough memory for array c\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double) * (6*nx) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

f = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( f == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}

u = ( double * )malloc((size_t)( sizeof(double) * m ));
if( u == NULL )
{
    printf( "no enough memory for array u\n" );
    return -1;
}

s = ( double * )malloc((size_t)( sizeof(double) * m ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}
printf( "\tn   = %6d\n", nx  );
printf( "\tm   = %6d\n", m   );
printf( "\tisw = %6d\n", isw );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &f[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &u[i] );
}

printf( "\n\tCoordinates (x,y)\n\n" );
printf( "\t     i     x[i]    yd[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d %8.3g %8.3g\n", i,x[i],f[i] );
}

printf( "\n" );
printf( "\tSpecified Points\n\n" );
printf( "\t     j    xl[j]\n");
for( j=0 ; j<m ; j++ )
{
    printf( "\t%6d %8.3g\n", j,u[j] );
}

fclose( fp );

ierr = ASL_dgissc(x, f, nx, u, s, m, y, c, isw, wk);

printf( "\n     ** Output **\n\n" );
```

```
    printf( "\tierr = %6d\n", ierr );

    printf( "\n" );
    for( j=0 ; j<m ; j++ )
    {
        printf( "\tfl[%6d]=%8.3g\n", j,s[j] );
    }

    printf( "\n" );
    for( j=0 ; j<nx ; j++ )
    {
        printf( "\ty[%6d]=%8.3g\n", j,y[j] );
    }

    printf( "\n" );
    printf( "\tc[i,j]\n\n" );
    printf( "\t                i=0      i=1      i=2\n" );
    for( j=0 ; j<nx-1 ; j++ )
    {
        printf( "\t j=%6d",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( " %8.3g", c[i+3*j] );
        }
        printf( "\n" );
    }

    free( x );
    free( f );
    free( u );
    free( s );
    free( y );
    free( c );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgissc ***

 ** Input **

n   =     25
m   =     10
isw =      2

Coordinates (x,y)

     i     x[i]    yd[i]
     0        0   0.0481
     1   0.0416    0.147
     2   0.0833    0.442
     3    0.125    0.579
     4    0.167    0.641
     5    0.208     0.74
     6     0.25    0.726
     7    0.292     1.09
     8    0.333    0.804
     9    0.375     1.02
    10    0.417    0.851
    11    0.458        1
    12      0.5    0.517
    13    0.542    0.692
    14    0.583    0.348
    15    0.625    0.385
    16    0.667   -0.102
    17    0.708   -0.135
    18     0.75   -0.339
    19    0.792   -0.571
    20    0.833   -0.525
    21    0.875   -0.841
    22    0.917   -0.795
    23    0.958    -1.08
    24        1    -1.18

Specified Points

     j    xl[j]
     0        0
     1      0.1
     2      0.2
     3      0.3
     4      0.4
     5      0.5
     6      0.6
     7      0.7
     8      0.8
     9      0.9
```

```
 ** Output **

ierr =      0

fl[     0]=   0.095
fl[     1]=   0.447
fl[     2]=   0.732
fl[     3]=   0.901
fl[     4]=   0.896
fl[     5]=   0.695
fl[     6]=   0.341
fl[     7]= -0.0861
fl[     8]=   -0.49
fl[     9]=  -0.849

y[     0]=   0.095
y[     1]=   0.245
y[     2]=   0.391
y[     3]=   0.527
y[     4]=   0.648
y[     5]=   0.751
y[     6]=   0.834
y[     7]=   0.893
y[     8]=    0.92
y[     9]=   0.915
y[    10]=   0.876
y[    11]=   0.802
y[    12]=   0.695
y[    13]=   0.563
y[    14]=   0.408
y[    15]=   0.237
y[    16]=  0.0578
y[    17]=  -0.121
y[    18]=  -0.294
y[    19]=  -0.458
y[    20]=  -0.613
y[    21]=  -0.761
y[    22]=  -0.907
y[    23]=   -1.05
y[    24]=    -1.2

c[i,j]

                i=0       i=1       i=2
 j=     0       3.6 2.13e-14      -5.9
 j=     1      3.57    -0.737    -18.1
 j=     2      3.41        -3    -11.7
 j=     3       3.1     -4.47    -5.13
 j=     4       2.7     -5.11    -5.98
 j=     5      2.25     -5.86    -7.46
 j=     6      1.72     -6.79    -21.1
 j=     7      1.05     -9.42     3.88
 j=     8      0.28     -8.94    -10.6
 j=     9     -0.52     -10.3     2.62
 j=    10     -1.36     -9.94    -0.59
 j=    11     -2.19       -10     24.5
 j=    12      -2.9     -6.95      2.1
 j=    13     -3.47     -6.69     18.3
 j=    14     -3.93      -4.4     10.8
 j=    15     -4.24     -3.05     29.4
 j=    16     -4.34     0.628     9.28
 j=    17     -4.24      1.79     7.52
 j=    18     -4.05      2.73     1.89
 j=    19     -3.81      2.96    -12.3
 j=    20     -3.63      1.42    -1.37
 j=    21     -3.52      1.25    -11.3
 j=    22     -3.47    -0.161     2.67
 j=    23     -3.47     0.173    -1.39
```

6.2.3 ASL_dgismc, ASL_rgismc
Least Squares Interpolation Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgismc or ASL_rgismc obtains least squares approximation cubic spline coefficients and calculates interpolation values at specified points. In addition optimum knot positions are obtained.

(2) **Usage**

Double precision:

```
ierr = ASL_dgismc  (x, yd, n, xk, nxk, &itmx, xl, fl, m, &s, y, c, iwk, wk1, wk2);
```

Single precision:

```
ierr = ASL_rgismc  (x, yd, n, xk, nxk, &itmx, xl, fl, m, &s, y, c, iwk, wk1, wk2);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xk	{D∗ / R∗}	nxk	Input	Initial estimates of knot positions xk[i − 1] < xk[i], i = 1, ··· , nxk − 1 xk[0] ≤ x[0] xk[nxk − 1] ≥ x[n − 1]
				Output	Optimum knot positions
5	nxk	I	1	Input	Number of knots
6	itmx	I*	1	Input	Maximum number of iterations (15 is suitable)
				Output	Actual iteration count
7	xl	{D∗ / R∗}	m	Input	Abscissa values of points where interpolation values are calculated.
8	fl	{D∗ / R∗}	m	Output	Least squares approximation cubic spline interpolation values at xl[i − 1]
9	m	I	1	Input	Number of interpolation values

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	s	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	1	Output	Least squares error of cubic spline approximation
11	y	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	nxk	Output	0th order terms of cubic spline coefficients
12	c	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): $c[(k-1)+3\times(j-1)]$ The value of cubic spline function f(t) at abscissa value t ($xk[j-1] \le t < xk[j]$): $f(t) = ((c[2+3\times(j-1)]\times d + c[1+3\times(j-1)])\times d + c[3\times(j-1)])\times d + y[j-1]$ where: $d = t - xk[j-1]$ **Size**: $3\times(nxk-1)$
13	iwk	I*	75	Work	Work area
14	wk1	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $n\times(nxk+6)$
15	wk2	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	3811	Work	Work area
16	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \ge 2$, $2 \le nxk \le 28$, and $itmx \le 20$

(b) $x[0] < x[1] < \cdots < x[n-1]$ (Ascending order)

(c) Sample points are distributed throughout the range determined by the end point knots.

(d) $xk[0] < xk[1] < \cdots < xk[nxk-1]$ (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	$xl[i-1]$ was outside the interpolation interval range ($xl[i-1] < x[0]$ or $xl[i-1] > x[n-1]$).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
1500	Calculations ended after itmx iterations without obtaining the minimum least squares error (s). (The data is nearly uncorrelated.)	Interpolation is performed using the current spline coefficients and least squares error.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
3030	Restriction (d) was not satisfied.	

(6) **Notes**

(a) Different cubic spline coefficients may be obtained and convergence status may be changed depending on the initial estimates of the knot positions.

(b) Usually values y[0] and y[nxk − 1] become extrapolated ones using cubic spline coefficients.

(c) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\\text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\\text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\\text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array xk, y, c and variable nxk must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation

$$y_i = \begin{cases} 1.0 - x_i & (0.0 \le x_i \le 0.5) \\ x_i & (0.5 < x_i \le 1.0) \\ 2.0 - x_i & (1.0 < x_i \le 2.0) \end{cases}$$

to find y_i at each point $x_i = 0.1 \times (i-1)$ $(i = 1, 2, \cdots, 21)$ and perform least squares approximation cubic spline interpolation with four knots $\{\xi_j\} = \{0.0, 0.33, 1.33, 2.0\}$ using these points as sample points. In addition, obtain the value of the cubic spline function at $xl_j = 0.25 \times k (k = 1, 2, \cdots, 7)$.

(b) Input data
x[i − 1]=x_i, yd[i − 1]=y_i (i = 1, ⋯, n), xk[j − 1]=ξ_j (j = 1, ⋯, nxk), xl[j − 1]=xl_j (k = 1, ⋯, m),
n = 21, nxk = 4, m = 7 and itmx = 15.

(c) Main program

```
/*      C interface example for ASL_dgismc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *f;
    int nx;
    double *xk;
    int nxk;
    int  itmx;
    double *u;
    double *s;
    int m;
    double er;
    double *y;
    double *c;
    int iwk[75];
    double *wk1;
    double wk2[3811];
    int ierr;
    int i,j,k,nwk;
    FILE *fp;

    fp = fopen( "dgismc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgismc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=21;
    nxk=4;
    m=7;
    itmx=15;
```

```
c = ( double * )malloc((size_t)( sizeof(double) * (3*(nxk-1)) ));
if( c == NULL )
{
    printf( "no enough memory for array c\n" );
    return -1;
}

nwk=nx*(nxk+6);
wk1 = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk1 == NULL )
{
    printf( "no enough memory for array wk1\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

f = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( f == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}

xk = ( double * )malloc((size_t)( sizeof(double) * nxk ));
if( xk == NULL )
{
    printf( "no enough memory for array xk\n" );
    return -1;
}

u = ( double * )malloc((size_t)( sizeof(double) * m ));
if( u == NULL )
{
    printf( "no enough memory for array u\n" );
    return -1;
}

s = ( double * )malloc((size_t)( sizeof(double) * m ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nxk ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

printf( "\tn   = %6d\n", nx  );
printf( "\tnxk = %6d\n", nxk );
printf( "\tm   = %6d\n", m   );
printf( "\titmx= %6d\n", itmx);
for( i=0 ; i<nxk ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &u[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &f[i] );
}

printf( "\n\tCoordinates (x,y)\n\n" );
printf( "\t     i     x[i]    yd[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d %8.3g %8.3g\n", i,x[i],f[i] );
}

printf( "\n" );
printf( "\tSpecified Points\n\n" );
printf( "\t     j    xl[j]\n");
for( j=0 ; j<m ; j++ )
{
```

```
        printf( "\t%6d %8.3g\n", j,u[j] );
    }

    printf( "\n" );
    printf( "\tLocations of Knots\n\n" );
    printf( "\t     k    xk[k]\n");
    for( k=0 ; k<nxk ; k++ )
    {
        printf( "\t%6d %8.3g\n", k,xk[k] );
    }

    fclose( fp );

    ierr = ASL_dgismc(x, f, nx, xk, nxk, &itmx, u, s, m, &er, y, c, iwk, wk1, wk2);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    for( j=0 ; j<m ; j++ )
    {
        printf( "\tfl[%6d]=%8.3g\n", j,s[j] );
    }
    printf( "\n" );

    for( j=0 ; j<nxk ; j++ )
    {
        printf( "\ty[%6d]=%8.3g\n", j,y[j] );
    }

    printf( "\n" );
    printf( "\tc[i,j]\n" );
    printf( "\t                i=0      i=1      i=2\n" );
    for( j=0 ; j<nxk-1 ; j++ )
    {
        printf( "\t j=%6d",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( " %8.3g", c[i+3*j] );
        }
        printf( "\n" );
    }
    printf( "\n" );
    printf( "\tOptimal Location of Knots\n\n" );
    for( k=0 ; k<nxk ; k++ )
    {
        printf( "\txk[%6d]= %8.3g\n", k,xk[k] );
    }
    printf( "\n" );
    printf( "\tNumber of Iterations\n\n" );
    printf( "\t  itmx=%6d\n",itmx );

    printf( "\n" );
    printf( "\tLeast Squares error\n\n" );
    printf( "\t  s= %8.3g\n", er );

    free( x );
    free( f );
    free( xk );
    free( u );
    free( s );
    free( y );
    free( c );
    free( wk1 );

    return 0;
}
```

(d) Output results

```
*** ASL_dgismc ***

 ** Input **

n   =     21
nxk =      4
m   =      7
itmx=     15

Coordinates (x,y)

     i     x[i]    yd[i]
     0        0        1
     1      0.1      0.9
     2      0.2      0.8
     3      0.3      0.7
     4      0.4      0.6
     5      0.5      0.5
     6      0.6      0.6
```

```
     7      0.7      0.7
     8      0.8      0.8
     9      0.9      0.9
    10        1        1
    11      1.1      0.9
    12      1.2      0.8
    13      1.3      0.7
    14      1.4      0.6
    15      1.5      0.5
    16      1.6      0.4
    17      1.7      0.3
    18      1.8      0.2
    19      1.9      0.1
    20        2        0

Specified Points

     j    xl[j]
     0     0.25
     1      0.5
     2     0.75
     3        1
     4     1.25
     5      1.5
     6     1.75

Locations of Knots

     k    xk[k]
     0        0
     1     0.33
     2     1.33
     3        2

 ** Output **

ierr =      0

fl[     0]=   0.743
fl[     1]=   0.538
fl[     2]=   0.773
fl[     3]=   0.916
fl[     4]=   0.786
fl[     5]=    0.51
fl[     6]=   0.221

y[     0]=   0.984
y[     1]=   0.558
y[     2]=   0.801
y[     3]=  0.0485

c[i,j]
                i=0      i=1      i=2
 j=     0   -0.345    -3.86     5.53
 j=     1    0.648      5.6    -14.3
 j=     2     1.15    -3.14     1.39

Optimal Location of Knots

xk[     0]=        0
xk[     1]=     0.57
xk[     2]=    0.774
xk[     3]=        2

Number of Iterations

  itmx=     3

Least Squares error

  s=   0.0289
```

6.2.4 ASL_dgidpc, ASL_rgidpc Derivative Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgidpc or ASL_rgidpc obtains cubic spline coefficients for "not-a-knot" endpoint conditions when endpoint conditions need not be entered and calculates first and second derivatives at specified points. The knots are set equal to sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgidpc  (x, y, n, xl, dl, ddl, m, c);
```

Single precision:

```
ierr = ASL_rgidpc  (x, y, n, xl, dl, ddl, m, c);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
				Output	0th order terms of cubic spline coefficients (Same as input values)
3	n	I	1	Input	Number of sample points
4	xl	{D∗ / R∗}	m	Input	Abscissa values of points where derivatives of cubic spline function are calculated
5	dl	{D∗ / R∗}	m	Output	First derivatives of cubic spline function at xl[i − 1] dl[i − 1] = (3.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)] where: x[j − 1] ≤ xl[i − 1] < x[j] d = xl[i − 1] − x[j − 1]
6	ddl	{D∗ / R∗}	m	Output	Second derivatives of cubic spline function at xl[i − 1] ddl[i − 1] = 6.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)] where: x[j − 1] ≤ xl[i − 1] < x[j] d = xl[i − 1] − x[j − 1]
7	m	I	1	Input	Number of points where derivatives are calculated
8	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

(a) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\\text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\\text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\\text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array x, y, c and variable n must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation

$$y_i = x_i e^{-4.0x_i}$$

to find y_i $(i = 1, 2, \cdots, 9)$ at each point $\{x_i\} = \{0.0, 0.1, 0.23, 0.34, 0.47, 0.59, 0.73, 0.92, 1.0\}$ and perform cubic spline interpolation using these points as sample points. In addition, obtain the first and second derivatives of the cubic spline function at the uniformly spaced points $xl_j = 0.1\times(j-1)(j = 1, 2, \cdots, 10)$.

(b) Input data
x[i − 1]=x_i, y[i − 1]=y_i (i = 1, ⋯ , n), xl[j − 1]=xl_j (j = 1, ⋯ , m), n = 9 and m = 10.

(c) Main program

```
/*      C interface example for ASL_dgidpc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int nx;
    double *u;
    double *ds;
    double *dds;
    int m;
    double *c;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dgidpc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgidpc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=9;
    m=10;
```

```
c = ( double * )malloc((size_t)( sizeof(double) * (3*(nx-1)) ));
if( c == NULL )
{
    printf( "no enough memory for array c\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

u = ( double * )malloc((size_t)( sizeof(double) * m ));
if( u == NULL )
{
    printf( "no enough memory for array u\n" );
    return -1;
}

ds = ( double * )malloc((size_t)( sizeof(double) * m ));
if( ds == NULL )
{
    printf( "no enough memory for array ds\n" );
    return -1;
}

dds = ( double * )malloc((size_t)( sizeof(double) * m ));
if( dds == NULL )
{
    printf( "no enough memory for array dds\n" );
    return -1;
}

printf( "\tn   = %6d\n", nx  );
printf( "\tm   = %6d\n", m   );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &u[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &y[i] );
}

printf( "\n" );
printf( "\tCoordinates (X,Y)\n\n" );
printf( "\t     i               x[i]               y[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],y[i] );
}

printf( "\n" );
printf( "\tSpecified Points\n\n" );
printf( "\t     j              xl[j]\n");
for( j=0 ; j<m ; j++ )
{
    printf( "\t%6d           %8.3g\n", j,u[j] );
}

fclose( fp );

ierr = ASL_dgidpc(x, y, nx, u, ds, dds, m, c);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n" );

printf( "\tThe Value of The First Derivative\n\n" );
for( j=0 ; j<m ; j++ )
{
    printf( "\t   dl[%6d]=%8.3g\n", j,ds[j] );
}
printf( "\n" );

printf( "\tThe Value of The Second Derivative\n\n" );
for( j=0 ; j<m ; j++ )
```

```
    {
        printf( "\t  ddl[%6d]=%8.3g\n", j,dds[j] );
    }
    printf( "\n" );

    printf( "\tSpline Coefficients\n\n" );
    printf( "\t  c[i,j]\n" );
    printf( "\t\t\t       i=0               i=1               i=2\n" );
    for( j=0 ; j<nx-1 ; j++ )
    {
        printf( "\tj=%6d",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( "          %8.3g", c[i+3*j] );
        }
        printf( "\n" );
    }
    printf( "\n" );

    free( x );
    free( y );
    free( u );
    free( ds );
    free( dds );
    free( c );

    return 0;
}
```

(d) Output results

```
*** ASL_dgidpc ***

 ** Input **

n   =      9
m   =     10

Coordinates (X,Y)

     i                x[i]               y[i]
     0                   0                  0
     1                 0.1              0.067
     2                0.23             0.0917
     3                0.34             0.0873
     4                0.47             0.0717
     5                0.59             0.0557
     6                0.73             0.0394
     7                0.92             0.0232
     8                   1             0.0183

Specified Points

     j               xl[j]
     0                 0.1
     1                 0.2
     2                 0.3
     3                 0.4
     4                 0.5
     5                 0.6
     6                 0.7
     7                 0.8
     8                 0.9
     9                   1

 ** Output **

ierr =      0

The Value of The First Derivative

   dl[     0]=   0.414
   dl[     1]=  0.0842
   dl[     2]= -0.0595
   dl[     3]=  -0.122
   dl[     4]=  -0.135
   dl[     5]=  -0.127
   dl[     6]=  -0.109
   dl[     7]= -0.0898
   dl[     8]= -0.0714
   dl[     9]= -0.0547

The Value of The Second Derivative

  ddl[     0]=   -4.39
  ddl[     1]=    -2.2
  ddl[     2]=  -0.966
  ddl[     3]=  -0.358
  ddl[     4]=   0.015
  ddl[     5]=   0.151
  ddl[     6]=   0.191
  ddl[     7]=   0.191
  ddl[     8]=   0.175
```

```
  ddl[     9]=   0.159

Spline Coefficients

  c[i,j]
                i=0                 i=1                 i=2
j=     0              0.963              -3.29              3.66
j=     1              0.414               -2.2              3.66
j=     2             0.0281             -0.769              1.36
j=     3            -0.0917              -0.32             0.783
j=     4             -0.135            -0.0145             0.245
j=     5             -0.128             0.0736            0.0659
j=     6             -0.104              0.101           -0.0269
j=     7             -0.068              0.086           -0.0269
```

6.2.5 ASL_dgidsc, ASL_rgidsc
Smoothed Derivative Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgidsc or ASL_rgidsc automatically obtains the optimum smoothed spline coefficients and the first and second derivative at specified points. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgidsc  (x, yd, n, xl, dl, ddl, m, y, c, isw, wk);
```

Single precision:

```
ierr = ASL_rgidsc  (x, yd, n, xl, dl, ddl, m, y, c, isw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xl	{D∗ / R∗}	m	Input	Abscissa values of points where derivatives of cubic spline function are calculated
5	dl	{D∗ / R∗}	m	Output	First derivatives of cubic spline function at xl[i − 1] dl[i − 1] = (3.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)] where: x[j − 1] ≤ xl[i − 1] < x[j] d = xl[i − 1] − x[j − 1]
6	ddl	{D∗ / R∗}	m	Output	Second derivatives of cubic spline function at xl[i − 1] ddl[i − 1] = 6.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)] where: x[j − 1] ≤ xl[i − 1] < x[j] d = xl[i − 1] − x[j − 1]
7	m	I	1	Input	Number of points where derivatives are calculated
8	y	{D∗ / R∗}	n	Output	0th order terms of cubic spline coefficients
9	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	isw	I	1	Input	Processing switch isw=1: When the abscissa value spacing may not be uniform. When sample points are nearly uncorrelated, limit value n up to 20. isw=2: When the abscissa value spacing is uniform.
11	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area Size: $n \times (2 \times n + 4)$ (when isw=1) $6 \times n$ (when isw=2)
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq 4$

(b) $x[0] < x[1] < \cdots < x[n-1]$ (Ascending order)

(c) If isw=2, abscissa values must be uniformly spaced.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
4000	The minimum cross-validation value could not be found. (The data is uncorrelated.)	

(6) **Notes**

(a) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx} \\ \text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy} \\ \text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz} \\ \text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array x, y, c and variable n must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation

$$y_i = \sin(3\pi x_i/2) + e_i \ (e_i : \text{Uniform random number in the interval } [-0.2, 0.2])$$

to find values y_i at each point $x_i = (i-1)/24$ $(i = 1, 2, \cdots, 25)$ and perform smoothed cubic spline interpolation using these points as sample points. In addition, obtain the first and second derivatives of the cubic spline function at the uniformly spaced points $xl_j = 0.1 \times (j-1)(j = 1, 2, \cdots, 10)$.

(b) Input data

x[i − 1]=x_i, yd[i − 1]=y_i (i = $1, \cdots,$ n), xl[j − 1]=xl_j (j = $1, \cdots,$ m), n=25, isw=2 and m=10.

(c) Main program

```
/*      C interface example for ASL_dgidsc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *f;
    int nx;
    double *u;
    double *ds;
    double *dds;
    int m;
    double *y;
    double *c;
    int isw;
    double *wk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dgidsc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgidsc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=25;
    m=10;
    isw=2;

    c = ( double * )malloc((size_t)( sizeof(double) * (3*(nx-1)) ));
    if( c == NULL )
    {
        printf( "no enough memory for array c\n" );
        return -1;
    }

    wk = ( double * )malloc((size_t)( sizeof(double) * (6*nx) ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }

    f = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( f == NULL )
    {
        printf( "no enough memory for array f\n" );
        return -1;
    }

    u = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( u == NULL )
    {
        printf( "no enough memory for array u\n" );
        return -1;
    }

    ds = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( ds == NULL )
    {
        printf( "no enough memory for array ds\n" );
        return -1;
    }

    dds = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( dds == NULL )
    {
```

```
    printf( "no enough memory for array dds\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

printf( "\tn   = %6d\n", nx  );
printf( "\tm   = %6d\n", m   );
printf( "\tisw = %6d\n", isw );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &f[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &u[i] );
}

printf( "\n" );
printf( "\tCoordinates (x,yd)\n\n" );
printf( "\t     i               x[i]              yd[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],f[i] );
}

printf( "\n" );
printf( "\tSpecified Points\n\n" );
printf( "\t     j              xl[j]\n");
for( j=0 ; j<m ; j++ )
{
    printf( "\t%6d           %8.3g\n", j,u[j] );
}

fclose( fp );

ierr = ASL_dgidsc(x, f, nx, u, ds, dds, m, y, c, isw, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n" );

printf( "\tThe Value of The First Derivative\n\n" );
for( j=0 ; j<m ; j++ )
{
    printf( "\t   dl[%6d]=%8.3g\n", j,ds[j] );
}
printf( "\n" );

printf( "\tThe Value of The Second Derivative\n\n" );
for( j=0 ; j<m ; j++ )
{
    printf( "\t  ddl[%6d]=%8.3g\n", j,dds[j] );
}
printf( "\n" );

printf( "\tSpline Coefficients\n\n" );
for( j=0 ; j<nx ; j++ )
{
    printf( "\t    y[%6d]=%8.3g\n", j,y[j] );
}
printf( "\n" );

printf( "\tc[i,j]\n\n" );
printf( "\t\t\t       i=0               i=1               i=2\n" );
for( j=0 ; j<nx-1 ; j++ )
{
    printf( "\tj=%6d",j );
    for( i=0 ; i<3 ; i++ )
    {
        printf( "          %8.3g", c[i+3*j] );
    }
    printf( "\n" );
}
printf( "\n" );

free( c );
free( wk );
free( x );
```

```
    free( f );
    free( u );
    free( ds );
    free( dds );
    free( y );

    return 0;
}
```

(d) Output results

```
*** ASL_dgidsc ***

 ** Input **

n   =     25
m   =     10
isw =      2

Coordinates (x,yd)

     i                x[i]             yd[i]
     0                   0            0.0481
     1              0.0416             0.147
     2              0.0833             0.442
     3               0.125             0.579
     4               0.167             0.641
     5               0.208              0.74
     6                0.25             0.726
     7               0.292              1.09
     8               0.333             0.804
     9               0.375              1.02
    10               0.417             0.851
    11               0.458                 1
    12                 0.5             0.517
    13               0.542             0.692
    14               0.583             0.348
    15               0.625             0.385
    16               0.667            -0.102
    17               0.708            -0.135
    18                0.75            -0.339
    19               0.792            -0.571
    20               0.833            -0.525
    21               0.875            -0.841
    22               0.917            -0.795
    23               0.958             -1.08
    24                   1             -1.18

Specified Points

     j               xl[j]
     0                   0
     1                 0.1
     2                 0.2
     3                 0.3
     4                 0.4
     5                 0.5
     6                 0.6
     7                 0.7
     8                 0.8
     9                 0.9

 ** Output **

ierr =      0

The Value of The First Derivative

   dl[     0]=     3.6
   dl[     1]=     3.3
   dl[     2]=    2.34
   dl[     3]=   0.888
   dl[     4]=   -1.03
   dl[     5]=    -2.9
   dl[     6]=   -4.07
   dl[     7]=   -4.27
   dl[     8]=   -3.77
   dl[     9]=   -3.48

The Value of The Second Derivative

  ddl[     0]=4.26e-14
  ddl[     1]=   -7.18
  ddl[     2]=   -11.4
  ddl[     3]=   -18.6
  ddl[     4]=   -20.1
  ddl[     5]=   -13.9
  ddl[     6]=   -7.72
  ddl[     7]=    3.12
  ddl[     8]=    5.31
  ddl[     9]=   0.809

Spline Coefficients
```

```
    y[     0]=   0.095
    y[     1]=   0.245
    y[     2]=   0.391
    y[     3]=   0.527
    y[     4]=   0.648
    y[     5]=   0.751
    y[     6]=   0.834
    y[     7]=   0.893
    y[     8]=    0.92
    y[     9]=   0.915
    y[    10]=   0.876
    y[    11]=   0.802
    y[    12]=   0.695
    y[    13]=   0.563
    y[    14]=   0.408
    y[    15]=   0.237
    y[    16]=  0.0578
    y[    17]=  -0.121
    y[    18]=  -0.294
    y[    19]=  -0.458
    y[    20]=  -0.613
    y[    21]=  -0.761
    y[    22]=  -0.907
    y[    23]=   -1.05
    y[    24]=    -1.2
```

c[i,j]

	i=0	i=1	i=2
j= 0	3.6	2.13e-14	-5.9
j= 1	3.57	-0.737	-18.1
j= 2	3.41	-3	-11.7
j= 3	3.1	-4.47	-5.13
j= 4	2.7	-5.11	-5.98
j= 5	2.25	-5.86	-7.46
j= 6	1.72	-6.79	-21.1
j= 7	1.05	-9.42	3.88
j= 8	0.28	-8.94	-10.6
j= 9	-0.52	-10.3	2.62
j= 10	-1.36	-9.94	-0.59
j= 11	-2.19	-10	24.5
j= 12	-2.9	-6.95	2.1
j= 13	-3.47	-6.69	18.3
j= 14	-3.93	-4.4	10.8
j= 15	-4.24	-3.05	29.4
j= 16	-4.34	0.628	9.28
j= 17	-4.24	1.79	7.52
j= 18	-4.05	2.73	1.89
j= 19	-3.81	2.96	-12.3
j= 20	-3.63	1.42	-1.37
j= 21	-3.52	1.25	-11.3
j= 22	-3.47	-0.161	2.67
j= 23	-3.47	0.173	-1.39

6.2.6 ASL_dgidmc, ASL_rgidmc
Least Squares Method Derivative Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgidmc or ASL_rgidmc obtains least squares approximation cubic spline coefficients and calculates first and second derivatives at specified points. In addition optimum knot positions are obtained.

(2) **Usage**

Double precision:

```
ierr = ASL_dgidmc  (x, yd, n, xk, nxk, &itmx, xl, dl, ddl, m, &s, y, c, iwk, wk1, wk2);
```

Single precision:

```
ierr = ASL_rgidmc  (x, yd, n, xk, nxk, &itmx, xl, dl, ddl, m, &s, y, c, iwk, wk1, wk2);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xk	{D∗ / R∗}	nxk	Input	Initial estimates of knot positions xk[i − 1] < xk[i], i = 1, ··· , nxk − 1 xk[0] ≤ x[0] xk[nxk − 1] ≥ x[n − 1]
				Output	Optimum knot positions
5	nxk	I	1	Input	Number of knots
6	itmx	I*	1	Input	Maximum number of iterations (15 is suitable)
				Output	Actual iteration count
7	xl	{D∗ / R∗}	m	Input	Abscissa values of points where derivatives of cubic spline function are calculated

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	dl	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	m	Output	First derivatives of cubic spline function at xl[i − 1] dl[i − 1] = (3.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)] where: xk[j − 1] ≤ xl[i − 1] < xk[j] d = xl[i − 1] − xk[j − 1]
9	ddl	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	m	Output	Second derivatives of cubic spline function at xl[i − 1] ddl[i − 1] = 6.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)] where: xk[j − 1] ≤ xl[i − 1] < xk[j] d = xl[i − 1] − xk[j − 1]
10	m	I	1	Input	Number of points where derivatives are calculated
11	s	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	1	Output	Least squares error of cubic spline approximation
12	y	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	nxk	Output	0th order terms of cubic spline coefficients
13	c	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	See Contents	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1] **Size**: 3×(nxk − 1)
14	iwk	I*	75	Work	Work area
15	wk1	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	See Contents	Work	Work area **Size**: n × (nxk + 6)
16	wk2	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	3811	Work	Work area
17	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2, 2 ≤ nxk ≤ 28, and itmx ≤ 20

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(c) Sample points are distributed throughout the range determined by the endpoint knots.

(d) xk[0] < xk[1] < ··· < xk[nxk − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
1500	Calculations ended after itmx iterations without obtaining the minimum least squares error (s). (The data is nearly uncorrelated.)	Interpolation is performed using the current cubic spline coefficients and least squares error.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
3030	Restriction (d) was not satisfied.	

(6) **Notes**

(a) Different cubic spline coefficients may be obtained and convergence status may be changed depending on the initial estimates of the knot positions.

(b) Usually values y[0] and y[nxk − 1] become extrapolated ones using cubic spline coefficients.

(c) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\\text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\\text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\\text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array xk, y, c and variable nxk must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation

$$y_i = \begin{cases} 1.0 - x_i & (0.0 \le x_i \le 0.5) \\ x_i & (0.5 < x_i \le 1.0) \\ 2.0 - x_i & (1.0 < x_i \le 2.0) \end{cases}$$

to find y_i at each point $x_i = 0.1 \times (i-1)$ $(i = 1, 2, \cdots, 21)$ and perform least squares approximation cubic spline interpolation with four knots $\{\xi_j\} = \{0.0, 0.33, 1.33, 2.0\}$ using these points as sample points. In addition, obtain the first and second derivatives of the cubic spline function at the uniformly spaced points $xl_j = 0.25 \times k (k = 1, 2, \cdots, 7)$.

(b) Input data
x[i − 1] = x_i, yd[i − 1] = y_i (i = 1, ⋯ , n),
xk[j − 1] = ξ_j (j = 1, ⋯ , nxk),
xl[j − 1] = xl_j (k = 1, ⋯ , m),
n = 21, nxk = 4, m = 7 and itmx = 15.

(c) Main program

```
/*      C interface example for ASL_dgidmc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *f;
    int nx;
    double *xk;
    int nxk;
    int  itmx;
    double *u;
    double *ds;
    double *dds;
    int m;
    double  er;
    double *y;
    double *c;
    int iwk[75];
    double *wk1;
    double wk2[3811];
    int ierr;
    int i,j,k,nwk;
    FILE *fp;

    fp = fopen( "dgidmc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgidmc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=21;
    nxk=4;
    m=7;
    itmx=15;

    c = ( double * )malloc((size_t)( sizeof(double) * (3*(nxk-1)) ));
    if( c == NULL )
    {
        printf( "no enough memory for array c\n" );
        return -1;
    }

    nwk=nx*(nxk+6);
    wk1 = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk1 == NULL )
    {
        printf( "no enough memory for array wk1\n" );
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }

    f = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( f == NULL )
    {
        printf( "no enough memory for array f\n" );
        return -1;
    }

    xk = ( double * )malloc((size_t)( sizeof(double) * nxk ));
    if( xk == NULL )
    {
        printf( "no enough memory for array xk\n" );
        return -1;
    }

    u = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( u == NULL )
    {
        printf( "no enough memory for array u\n" );
        return -1;
    }

    ds = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( ds == NULL )
    {
        printf( "no enough memory for array ds\n" );
        return -1;
    }
```

```
dds = ( double * )malloc((size_t)( sizeof(double) * m ));
if( dds == NULL )
{
    printf( "no enough memory for array dds\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nxk ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

printf( "\tn   = %6d\n", nx  );
printf( "\tnxk = %6d\n", nxk );
printf( "\tm   = %6d\n", m   );
printf( "\titmx= %6d\n", itmx);
for( i=0 ; i<nxk ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
}
for( i=0 ; i<m ; i++ )
{
    fscanf( fp, "%lf", &u[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &f[i] );
}

printf( "\n\tCoordinates (x,yd)\n\n" );
printf( "\t     i      x[i]    yd[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d %8.3g %8.3g\n", i,x[i],f[i] );
}

printf( "\n" );
printf( "\tSpecified Points\n\n" );
printf( "\t     j     xl[j]\n");
for( j=0 ; j<m ; j++ )
{
    printf( "\t%6d %8.3g\n", j,u[j] );
}

printf( "\n" );
printf( "\tLocations of Knots\n\n" );
printf( "\t     k     xk[k]\n");
for( k=0 ; k<nxk ; k++ )
{
    printf( "\t%6d %8.3g\n", k,xk[k] );
}

fclose( fp );

ierr = ASL_dgidmc(x, f, nx, xk, nxk, &itmx, u, ds, dds, m, &er, y, c, iwk, wk1, wk2);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );

printf( "\n\tThe value of the first derivative\n\n" );
for( j=0 ; j<m ; j++ )
{
    printf( "\t   dl[%6d]=%8.3g\n", j,ds[j] );
}
printf( "\n" );

printf( "\n\tThe value of the second derivative\n\n" );
for( j=0 ; j<m ; j++ )
{
    printf( "\t  ddl[%6d]=%8.3g\n", j,dds[j] );
}
printf( "\n" );

for( j=0 ; j<nxk ; j++ )
{
    printf( "\t    y[%6d]=%8.3g\n", j,y[j] );
}

printf( "\n" );
printf( "\tc[i,j]\n" );
printf( "\t\t        i=0               i=1               i=2\n" );
for( j=0 ; j<nxk-1 ; j++ )
```

```
    {
        printf( "\tj=%6d    ",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( "%8.3g          ", c[i+3*j] );
        }
        printf( "\n" );
    }
    printf( "\n" );
    printf( "\tOptimal Location of Knots\n\n" );
    for( k=0 ; k<nxk ; k++ )
    {
        printf( "\t   xk[%6d]= %8.3g\n", k,xk[k] );
    }
    printf( "\n" );
    printf( "\tNumber of Iterations\n" );
    printf( "\t  itmx=%6d\n",itmx );

    printf( "\n" );
    printf( "\tLeast Squares error\n" );
    printf( "\t  s= %8.3g\n", er );
    free( x );
    free( f );
    free( xk );
    free( u );
    free( ds );
    free( dds );
    free( y );
    free( c );
    free( wk1 );

    return 0;
}
```

(d) Output results

```
*** ASL_dgidmc ***

 ** Input **

n   =     21
nxk =      4
m   =      7
itmx=     15

Coordinates (x,yd)

     i      x[i]    yd[i]
     0         0        1
     1       0.1      0.9
     2       0.2      0.8
     3       0.3      0.7
     4       0.4      0.6
     5       0.5      0.5
     6       0.6      0.6
     7       0.7      0.7
     8       0.8      0.8
     9       0.9      0.9
    10         1        1
    11       1.1      0.9
    12       1.2      0.8
    13       1.3      0.7
    14       1.4      0.6
    15       1.5      0.5
    16       1.6      0.4
    17       1.7      0.3
    18       1.8      0.2
    19       1.9      0.1
    20         2        0

Specified Points

     j     xl[j]
     0     0.25
     1      0.5
     2     0.75
     3        1
     4     1.25
     5      1.5
     6     1.75

Locations of Knots

     k     xk[k]
     0         0
     1     0.33
     2     1.33
     3         2

 ** Output **

ierr =      0
```

```
The value of the first derivative

   dl[     0]=   -1.24
   dl[     1]= -0.0544
   dl[     2]=    1.27
   dl[     3]= -0.0606
   dl[     4]=  -0.899
   dl[     5]=   -1.22
   dl[     6]=   -1.01


The value of the second derivative

  ddl[     0]=   0.582
  ddl[     1]=    8.88
  ddl[     2]=   -4.26
  ddl[     3]=   -4.39
  ddl[     4]=   -2.31
  ddl[     5]=  -0.225
  ddl[     6]=    1.86

    y[     0]=   0.984
    y[     1]=   0.558
    y[     2]=   0.801
    y[     3]=  0.0485

c[i,j]
            i=0               i=1               i=2
j=     0      -0.345            -3.86              5.53
j=     1       0.648              5.6             -14.3
j=     2        1.15            -3.14              1.39

Optimal Location of Knots

   xk[     0]=        0
   xk[     1]=     0.57
   xk[     2]=    0.774
   xk[     3]=        2

Number of Iterations
  itmx=     3

Least Squares error
  s=   0.0289
```

6.2.7 ASL_dgiipc, ASL_rgiipc
Integral Values and Cubic Spline Coefficients

(1) **Function**

ASL_dgiipc or ASL_rgiipc obtains cubic spline coefficients for "not-a-knot" endpoint conditions when endpoint conditions need not be entered and calculates the integral over the specified range. The knots are set equal to sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiipc  (x, y, n, a, b, &q, c);
```

Single precision:

```
ierr = ASL_rgiipc  (x, y, n, a, b, &q, c);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]) for i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
				Output	0th order terms of cubic spline coefficients (Same as input values)
3	n	I	1	Input	Number of sample points
4	a	{D / R}	1	Input	Lower limit of the integration range
5	b	{D / R}	1	Input	Upper limit of the integration range
6	q	{D∗ / R∗}	1	Output	Integral value of cubic spline function over abscissa interval [a, b]
7	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	a or b was outside the interpolation interval range.	A function extrapolated from the cubic spline coefficients at the endpoints is integrated.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

(a) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\\text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\\text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\\text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array x, y, c and variable n must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation $y_i = x_i e^{-4.0x_i}$ to find y_i $(i = 1, 2, \cdots, 9)$ at each point
$\{x_i\} = \{0.0, 0.1, 0.23, 0.34, 0.47, 0.59, 0.73, 0.92, 1.0\}$
and perform cubic spline interpolation using these points as sample points. In addition, obtain the integral value of the cubic spline function over abscissa interval $[0.2, 0.5]$.

(b) Input data
x[i − 1]=x_i, y[i − 1]=y_i (i = 1, · · · , n), a=0.2, b=0.5 and n=9.

(c) Main program

```
/*      C interface example for ASL_dgiipc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int nx;
    double a;
    double b;
    double q;
    double *c;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dgiipc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgiipc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=9;
    a=0.2;
    b=0.5;

    c = ( double * )malloc((size_t)( sizeof(double) * (3*(nx-1)) ));
    if( c == NULL )
    {
        printf( "no enough memory for array c\n" );
```

```
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }


    y = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    printf( "\tn = %6d\n", nx );
    printf( "\n\tLimits of Integration a=%8.3g\n", a );
    printf( "\t                       b=%8.3g\n", b );
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
    }
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
    }

    printf( "\n\tCoordinates (x,y)\n\n" );
    printf( "\t     i     x[i]     y[i]\n");
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t%6d %8.3g %8.3g\n", i,x[i],y[i] );
    }

    fclose( fp );

    ierr = ASL_dgiipc(x, y, nx, a, b, &q, c);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\tIntegral\n\n" );
    printf( "\t  q=%8.3g\n",q );

    printf( "\n" );
    printf( "\tSpline Coefficients\n\n" );
    printf( "\t  c[i,j]\n" );
    printf( "\t               i=0      i=1      i=2\n" );
    for( j=0 ; j<nx-1 ; j++ )
    {
        printf( "\t j=%6d",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( " %8.3g", c[i+3*j] );
        }
        printf( "\n" );
    }

    free( x );
    free( y );
    free( c );

    return 0;
}
```

(d) Output results

```
*** ASL_dgiipc ***

 ** Input **

n =      9

Limits of Integration a=     0.2
                      b=     0.5

Coordinates (x,y)

     i     x[i]     y[i]
     0        0        0
     1      0.1    0.067
     2     0.23   0.0917
     3     0.34   0.0873
     4     0.47   0.0717
     5     0.59   0.0557
     6     0.73   0.0394
```

```
     7     0.92   0.0232
     8        1   0.0183

 ** Output **

ierr =      0

Integral

  q=  0.0252

Spline Coefficients

  c[i,j]
              i=0       i=1       i=2
 j=     0    0.963    -3.29      3.66
 j=     1    0.414     -2.2      3.66
 j=     2   0.0281   -0.769      1.36
 j=     3  -0.0917    -0.32     0.783
 j=     4   -0.135  -0.0145     0.245
 j=     5   -0.128   0.0736    0.0659
 j=     6   -0.104    0.101   -0.0269
 j=     7   -0.068    0.086   -0.0269
```

6.2.8 ASL_dgiisc, ASL_rgiisc
Smoothed Integral Value and Cubic Spline Coefficients

(1) **Function**

ASL_dgiisc or ASL_rgiisc automatically obtains the optimum smoothed spline coefficients and calculates the value of the integral over the specified range. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiisc  (x, yd, n, a, b, &q, y, c, isw, wk);
```

Single precision:

```
ierr = ASL_rgiisc  (x, yd, n, a, b, &q, y, c, isw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	a	{D / R}	1	Input	Lower limit of the integration range
5	b	{D / R}	1	Input	Upper limit of the integration range
6	q	{D∗ / R∗}	1	Output	Integral value of cubic spline function over abscissa interval [a, b]
7	y	{D∗ / R∗}	n	Output	0th order terms of cubic spline coefficients
8	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
9	isw	I	1	Input	Processing switch isw=1: When the abscissa value spacing may not be uniform. When sample points are nearly uncorrelated, limit value n up to 20. isw=2: When the abscissa value spacing is uniform.
10	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: n × (2 × n + 4) (when isw=1) 6 × n (when isw=2)
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n $\geq$ 4

(b) x[0] $<$ x[1] $<\cdots<$ x[n $-$ 1] (Ascending order)

(c) If isw=2, abscissa values must be uniformly spaced.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	a or b was outside the interpolation interval range.	A function extrapolated from the cubic spline coefficients at the endpoints is integrated.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
4000	The minimum cross-validation value could not be found. (The data is uncorrelated.)	

(6) **Notes**

(a) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\ \text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\ \text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\ \text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array x, y, c and variable n must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation $y_i = \sin(3\pi x_i/2) + e_i$ (e_i:Uniform random number in the interval $[-0.2, 0.2]$) to find values y_i at each point $x_i = (i-1)/24$ ($i = 1, 2, \cdots, 25$) and perform smoothed cubic spline interpolation using these points as sample points. In addition, obtain the integral value of the cubic spline function over abscissa interval $[0.2, 0.5]$.

(b) Input data
x[i $-$ 1] = x_i, yd[i $-$ 1] = y_i (i = 1, $\cdots$, n), a = 0.2, b = 0.5, n = 25, isw = 2 and m = 10.

(c) Main program

```
/*      C interface example for ASL_dgiisc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *f;
    int nx;
    double a;
    double b;
    double q;
    double *y;
    double *c;
    int isw;
```

```
double *wk;
int ierr;
int i,j;
FILE *fp;

fp = fopen( "dgiisc.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dgiisc ***\n" );
printf( "\n     ** Input **\n\n" );
nx=25;
isw=2;
a=0.2;
b=0.5;

c = ( double * )malloc((size_t)( sizeof(double) * (3*(nx-1)) ));
if( c == NULL )
{
    printf( "no enough memory for array c\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double) * (6*nx) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}


f = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( f == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}


y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

printf( "\tn  = %6d\n", nx );
printf( "\tisw= %6d\n", isw );
printf( "\n\tLimits of Integration a=%8.3g\n", a );
printf( "\t                      b=%8.3g\n", b );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &f[i] );
}

printf( "\n\tCoordinates (x,yd)\n\n" );
printf( "\t     i     x[i]    yd[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d %8.3g %8.3g\n", i,x[i],f[i] );
}

fclose( fp );

ierr = ASL_dgiisc(x, f, nx, a, b, &q, y, c, isw, wk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n" );

printf( "\tIntegral" );
printf( "\tq=%8.3g\n",q );

printf( "\n" );
printf( "\tSpline Coefficients\n\n" );
for( i=0 ; i<nx ; i++ )
{
```

```
        printf( "\t  y[%6d]=%8.3g\n", i,y[i] );
    }
    printf( "\n" );
    printf( "\tc[i,j]\n\n" );
    printf( "\t               i=0      i=1      i=2\n" );
    for( j=0 ; j<nx-1 ; j++ )
    {
        printf( "\t j=%6d",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( " %8.3g", c[i+3*j] );
        }
        printf( "\n" );
    }

    free( x );
    free( f );
    free( y );
    free( c );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgiisc ***

 ** Input **

n  =     25
isw=      2

Limits of Integration a=     0.2
                      b=     0.5

Coordinates (x,yd)

     i     x[i]    yd[i]
     0        0   0.0481
     1   0.0416    0.147
     2   0.0833    0.442
     3    0.125    0.579
     4    0.167    0.641
     5    0.208     0.74
     6     0.25    0.726
     7    0.292     1.09
     8    0.333    0.804
     9    0.375     1.02
    10    0.417    0.851
    11    0.458        1
    12      0.5    0.517
    13    0.542    0.692
    14    0.583    0.348
    15    0.625    0.385
    16    0.667   -0.102
    17    0.708   -0.135
    18     0.75   -0.339
    19    0.792   -0.571
    20    0.833   -0.525
    21    0.875   -0.841
    22    0.917   -0.795
    23    0.958    -1.08
    24        1    -1.18

 ** Output **

ierr =      0

Integral    q=   0.255

Spline Coefficients

  y[     0]=   0.095
  y[     1]=   0.245
  y[     2]=   0.391
  y[     3]=   0.527
  y[     4]=   0.648
  y[     5]=   0.751
  y[     6]=   0.834
  y[     7]=   0.893
  y[     8]=    0.92
  y[     9]=   0.915
  y[    10]=   0.876
  y[    11]=   0.802
  y[    12]=   0.695
  y[    13]=   0.563
  y[    14]=   0.408
  y[    15]=   0.237
```

```
  y[    16]=  0.0578
  y[    17]=  -0.121
  y[    18]=  -0.294
  y[    19]=  -0.458
  y[    20]=  -0.613
  y[    21]=  -0.761
  y[    22]=  -0.907
  y[    23]=   -1.05
  y[    24]=    -1.2

c[i,j]

                i=0      i=1      i=2
 j=     0      3.6 2.13e-14     -5.9
 j=     1     3.57   -0.737    -18.1
 j=     2     3.41       -3    -11.7
 j=     3      3.1    -4.47    -5.13
 j=     4      2.7    -5.11    -5.98
 j=     5     2.25    -5.86    -7.46
 j=     6     1.72    -6.79    -21.1
 j=     7     1.05    -9.42     3.88
 j=     8     0.28    -8.94    -10.6
 j=     9    -0.52    -10.3     2.62
 j=    10    -1.36    -9.94    -0.59
 j=    11    -2.19      -10     24.5
 j=    12     -2.9    -6.95      2.1
 j=    13    -3.47    -6.69     18.3
 j=    14    -3.93     -4.4     10.8
 j=    15    -4.24    -3.05     29.4
 j=    16    -4.34    0.628     9.28
 j=    17    -4.24     1.79     7.52
 j=    18    -4.05     2.73     1.89
 j=    19    -3.81     2.96    -12.3
 j=    20    -3.63     1.42    -1.37
 j=    21    -3.52     1.25    -11.3
 j=    22    -3.47   -0.161     2.67
 j=    23    -3.47    0.173    -1.39
```

6.2.9 ASL_dgiimc, ASL_rgiimc
Least Squares Method Integral Value and Cubic Spline Coefficients

(1) **Function**

ASL_dgiimc or ASL_rgiimc obtains least squares approximation cubic spline coefficients and calculates the value of the integral over the specified range. In addition optimum knot positions are obtained.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiimc  (x, yd, n, xk, nxk, &itmx, a, b, &q, &s, y, c, iwk, wk1, wk2);
```

Single precision:

```
ierr = ASL_rgiimc  (x, yd, n, xk, nxk, &itmx, a, b, &q, &s, y, c, iwk, wk1, wk2);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xk	{D∗ / R∗}	nxk	Input	Initial estimates of knot positions xk[i − 1] < xk[i], i = 1, ··· , nxk − 1 xk[0] ≤ x[0] xk[nxk − 1] ≥ x[n − 1]
				Output	Optimum knot positions
5	nxk	I	1	Input	Number of knots
6	itmx	I*	1	Input	Maximum number of iterations (15 is suitable)
				Output	Actual iteration count
7	a	{D / R}	1	Input	Lower limit of the integration range
8	b	{D / R}	1	Input	Upper limit of the integration range
9	q	{D∗ / R∗}	1	Output	Integral value of cubic spline function over abscissa interval [a, b]

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	s	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	1	Output	Least squares error of cubic spline approximation
11	y	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	nxk	Output	0th order terms of cubic spline coefficients
12	c	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1] **Size**: 3×(nxk − 1)
13	iwk	I*	75	Work	Work area
14	wk1	I*	See Contents	Work	Work area **Size**: n × (nxk + 6)
15	wk2	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	3811	Work	Work area
16	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2, 2 ≤ nxk ≤ 28, and itmx ≤ 20

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(c) Sample points are distributed throughout the range determined by the endpoint knots.

(d) xk[0] < xk[1] < ··· < xk[nxk − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	a or b was outside the interpolation interval range.	A function extrapolated from the cubic spline coefficients at the endpoints is integrated.
1500	Calculations ended after itmx iterations without obtaining the minimum least squares error (s). (The data is nearly uncorrelated.)	Interpolation is performed using the current spline coefficients and least squares error.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
3030	Restriction (d) was not satisfied.	

(6) **Notes**

(a) Different cubic spline coefficients may be obtained and convergence status may be changed depending on the initial estimates of the knot positions.

(b) Usually values y[0] and y[nxk − 1] become extrapolated ones using cubic spline coefficients.

(c) To continue further obtaining interpolation values, derivatives or integrals, you have called this function and then call function 6.2.18 $\left\{\begin{matrix}\text{ASL\_dgiscx}\\ \text{ASL\_rgiscx}\end{matrix}\right\}$, 6.2.19 $\left\{\begin{matrix}\text{ASL\_dgidcy}\\ \text{ASL\_rgidcy}\end{matrix}\right\}$ or 6.2.20 $\left\{\begin{matrix}\text{ASL\_dgiicz}\\ \text{ASL\_rgiicz}\end{matrix}\right\}$, respectively. In this case, contents of array xk, y, c and variable nxk must input to corresponding arguments of the succeeding function. Obtaining derivatives, there is no need to pass array y. This enables you to eliminate unnecessary calculations since you calculate cubic spline coefficients only once.

(7) **Example**

(a) Problem
Use the equation

$$y_i = \begin{cases} 1.0 - x_i & (0.0 \le x_i \le 0.5) \\ x_i & (0.5 < x_i \le 1.0) \\ 2.0 - x_i & (1.0 < x_i \le 2.0) \end{cases}$$

to find y_i at each point $x_i = 0.1 \times (i-1)$ $(i = 1, 2, \cdots, 21)$ and perform least squares approximation cubic spline interpolation with four knots $\{\xi_j\} = \{0.0, 0.33, 1.33, 2.0\}$ using these points as sample points. In addition, obtain the integral value of the cubic spline function over abscissa interval $[0.5, 1.5]$.

(b) Input data
x[i − 1]=x_i, yd[i − 1]=y_i (i = 1, ⋯, n), xk[j − 1]=ξ_j (j = 1, ⋯, nxk),
a=0.5, b=1.5, n=21, nxk=4 and itmx=15.

(c) Main program

```
/*      C interface example for ASL_dgiimc */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *f;
    int nx;
    double *xk;
    int nxk;
    int itmx;
    double a;
    double b;
    double q;
    double er;
    double *y;
    double *c;
    int iwk[75];
    double *wk1;
    double wk2[3811];
    int ierr;
    int i,j,k,nwk;
    FILE *fp;

    fp = fopen( "dgiimc.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgiimc ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=21;
    nxk=4;
    a=0.5;
    b=1.5;
```

```
itmx=15;

c = ( double * )malloc((size_t)( sizeof(double) * (3*(nxk-1)) ));
if( c == NULL )
{
    printf( "no enough memory for array c\n" );
    return -1;
}

nwk=nx*(nxk+6);
wk1 = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk1 == NULL )
{
    printf( "no enough memory for array wk1\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

f = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( f == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}

xk = ( double * )malloc((size_t)( sizeof(double) * nxk ));
if( xk == NULL )
{
    printf( "no enough memory for array xk\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nxk ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

printf( "\tn   = %6d\n", nx  );
printf( "\tnxk = %6d\n", nxk );
printf( "\titmx= %6d\n", itmx);
printf( "\n\tLimits of Integration   a=%8.3g\n", a);
printf( "\t                        b=%8.3g\n", b);
for( i=0 ; i<nxk ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &f[i] );
}

printf( "\n\tCoordinates (x,yd)\n\n" );
printf( "\t     i     x[i]    yd[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d %8.3g %8.3g\n", i,x[i],f[i] );
}

printf( "\n" );
printf( "\tLocations of Knots\n\n" );
printf( "\t     k    xk[k]\n");
for( k=0 ; k<nxk ; k++ )
{
    printf( "\t%6d %8.3g\n", k,xk[k] );
}

fclose( fp );

ierr = ASL_dgiimc(x, f, nx, xk, nxk, &itmx, a, b, &q, &er, y, c, iwk, wk1, wk2);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n" );

printf( "\tIntegral\n\n" );
printf( "\t  q=%8.3g\n",q );
printf( "\n" );
```

```
    for( j=0 ; j<nxk ; j++ )
    {
        printf( "\ty[%6d]=%8.3g\n", j,y[j] );
    }

    printf( "\n" );
    printf( "\tc[i,j]\n\n" );
    printf( "\t               i=0      i=1      i=2\n" );
    for( j=0 ; j<nxk-1 ; j++ )
    {
        printf( "\t j=%6d",j );
        for( i=0 ; i<3 ; i++ )
        {
            printf( " %8.3g", c[i+3*j] );
        }
        printf( "\n" );
    }
    printf( "\n" );
    printf( "\tOptimal Location of Knots\n\n" );
    for( k=0 ; k<nxk ; k++ )
    {
        printf( "\t  xk[%6d]= %8.3g\n", k,xk[k] );
    }
    printf( "\n" );
    printf( "\tNumber of Iterations\n\n" );
    printf( "\t  itmx=%6d\n",itmx );

    printf( "\n" );
    printf( "\tLeast Squares Error\n\n" );
    printf( "\t  s= %8.3g\n", er );

    free( x );
    free( f );
    free( xk );
    free( y );
    free( c );
    free( wk1 );

    return 0;
}
```

(d) Output results

```
*** ASL_dgiimc ***

 ** Input **

n   =     21
nxk =      4
itmx=     15

Limits of Integration   a=     0.5
                        b=     1.5

Coordinates (x,yd)

     i     x[i]    yd[i]
     0        0        1
     1      0.1      0.9
     2      0.2      0.8
     3      0.3      0.7
     4      0.4      0.6
     5      0.5      0.5
     6      0.6      0.6
     7      0.7      0.7
     8      0.8      0.8
     9      0.9      0.9
    10        1        1
    11      1.1      0.9
    12      1.2      0.8
    13      1.3      0.7
    14      1.4      0.6
    15      1.5      0.5
    16      1.6      0.4
    17      1.7      0.3
    18      1.8      0.2
    19      1.9      0.1
    20        2        0

Locations of Knots

     k    xk[k]
     0        0
     1     0.33
     2     1.33
     3        2

 ** Output **

ierr =      0
```

```
Integral

  q=   0.755

y[     0]=   0.984
y[     1]=   0.558
y[     2]=   0.801
y[     3]=  0.0485

c[i,j]

                i=0      i=1      i=2
 j=     0   -0.345    -3.86     5.53
 j=     1    0.648      5.6    -14.3
 j=     2     1.15    -3.14     1.39

Optimal Location of Knots

  xk[     0]=        0
  xk[     1]=     0.57
  xk[     2]=    0.774
  xk[     3]=        2

Number of Iterations

  itmx=     3

Least Squares Error

  s=   0.0289
```

6.2.10 ASL_dgiccp, ASL_rgiccp
Cubic Spline Coefficients (Endpoint Condition Input Unnecessary)

(1) **Function**

ASL_dgiccp or ASL_rgiccp obtains cubic spline coefficients for "not-a-knot" endpoint conditions when endpoint conditions need not be entered. The knots are set equal to sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiccp  (x, y, n, c);
```

Single precision:

```
ierr = ASL_rgiccp  (x, y, n, c);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]) for i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
				Output	0th order terms of cubic spline coefficients (Same as input values)
3	n	I	1	Input	Number of sample points
4	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
5	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

None

6.2.11 ASL_dgiccq, ASL_rgiccq
Cubic Spline Coefficients (Endpoint Conditions Are Input)

(1) **Function**

ASL_dgiccq or ASL_rgiccq obtains cubic spline coefficients for specified endpoint conditions. The knots are set equal to sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgiccq (x, y, n, end, c, isw);

Single precision:

ierr = ASL_rgiccq (x, y, n, end, c, isw);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]) for i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
				Output	0th order terms of cubic spline coefficients (Same as input values)
3	n	I	1	Input	Number of sample points
4	end	{D∗ / R∗}	4	Input	Endpoint conditions The following values must be set to array end correspond to the values of isw. (See the explanation of the algorithm or the notes.) **Endpoint conditions at initial point** x[0] isw[0]=1:end[0]=value of y' at initial point isw[0]=2:end[0]=value of y'' at initial point isw[0]=3:end[0]=value of y''' at initial point isw[0]=4:end[0]=λ_1, end[1]=d_1 **Endpoint conditions at final point** x[n − 1] isw[2]=1:end[2]=value of y' at final point isw[2]=2:end[2]=value of y'' at final point isw[2]=3:end[2]=value of y''' at final point isw[1]=4:end[2]=λ_{n-1}, end[3]=d_{n-1}

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
5	c	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	$3\times(\text{n}-1)$	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): $\text{c}[(\text{k}-1)+3\times(\text{j}-1)]$ The value of cubic spline function f(t) at abscissa value t ($\text{x}[\text{j}-1] \le \text{t} < \text{x}[\text{j}]$): $\text{f(t)} = ((\text{c}[2+3\times(\text{j}-1)]\times\text{d}+\text{c}[1+3\times(\text{j}-1)])\times\text{d}+\text{c}[3\times(\text{j}-1)])\times\text{d}+\text{y}[\text{j}-1]$ where: $\text{d}=\text{t}-\text{x}[\text{j}-1]$
6	isw	I*	2	Input	Switch for the value set in endpoint condition array end. isw[0] and isw[1] correspond to the initial and final point, respectively (See argument end).
7	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{n} \ge 2$

(b) $\text{x}[0] < \text{x}[1] < \cdots < \text{x}[\text{n}-1]$ (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

(a) You must declare the size of the array end to be 4 regardless of whether isw[0] or isw[1] equal to 1, 2 or 3. In this case, no initialization is necessary for end[1] and end[3].

(b) When isw[0]$\ge$ 4 or isw[1]$\ge$ 4, the values must be set in array end are the following values $\lambda_1, d_1, \mu_\text{n}$ and d_n of coefficient of simultaneous linear equations obtaining cubic spline coefficients. For details, see section 6.1.2Algorithms Used.

$$\begin{bmatrix} 2 & \lambda_1 & & & & & \\ & & & & & 0 & \\ \mu_2 & 2 & \lambda_2 & & & & \\ & \mu_3 & \ddots & \ddots & & & \\ & \ddots & \ddots & \lambda_{\text{n}-2} & & & \\ & \mu_{\text{n}-1} & & 2 & & \lambda_{\text{n}-1} & \\ 0 & & & \mu_\text{n} & & 2 & \end{bmatrix} \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ \vdots \\ \vdots \\ M_{\text{n}-1} \\ M_\text{n} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ \vdots \\ d_{\text{n}-1} \\ d_\text{n} \end{bmatrix}$$

where $M_1, \cdots, M_\text{n}$ is second derivatives of the cubic spline function.

6.2.12 ASL_dgiccr, ASL_rgiccr
Cubic Spline Coefficients (Periodic Spline)

(1) **Function**

ASL_dgiccr or ASL_rgiccr obtains cubic spline coefficients for periodic endpoint conditions. The knots are set equal to sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgiccr (x, y, n, c, wk);

Single precision:

ierr = ASL_rgiccr (x, y, n, c, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]) for i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
				Output	0th order terms of cubic spline coefficients (Same as input values)
3	n	I	1	Input	Number of sample points
4	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
5	wk	{D∗ / R∗}	5 × n	Work	Work area
6	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 4

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(c) y[0] = y[n − 1] (Periodic endpoint condition)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
2000	Restriction (c) was not satisfied.	Processing is performed regarding y[n − 1] as y[0].
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

(a) The periodic spline is interpolated as if the curved-line pattern on the abscissa interval $[x[0], x[n-1]]$ were extended continuously. So, only when values such as interpolation values, derivatives, and integrals are within the interpolation interval, the result makes its meaning.

6.2.13 ASL_dgiccs, ASL_rgiccs
Cubic Spline Coefficients (Automatic Smoothing)

(1) **Function**

ASL_dgiccs or ASL_rgiccs automatically determines the smoothing control variable and obtains the optimum smoothed cubic spline coefficients. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiccs  (x, yd, n, y, c, &s, isw, wk);
```

Single precision:

```
ierr = ASL_rgiccs  (x, yd, n, y, c, &s, isw, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	y	{D∗ / R∗}	n	Output	0th order terms of cubic spline coefficients
5	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
6	s	{D∗ / R∗}	1	Output	Sum of the squares of the error ($\sum_{i=1}^{n}\{yd[i-1]-y[i-1]\}^2$)
7	isw	I	1	Input	Processing switch isw=1: When the abscissa value spacing may not be uniform. When sample points are nearly uncorrelated, limit value n up to 20. isw=2: When the abscissa value spacing is uniform.
8	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: n × (2 × n + 4) (when isw=1) 6 × n (when isw=2)
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 4

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(c) If isw=2, abscissa values must be uniformly spaced.

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
4000	The minimum cross-validation value could not be found. (The data is uncorrelated.)	

(6) **Notes**

None

6.2.14 ASL_dgicco, ASL_rgicco
Cubic Spline Coefficients (Automatic Smoothing Periodic Conditions)

(1) **Function**

ASL_dgicco or ASL_rgicco automatically determines the smoothing control variable and obtains the optimum smoothed cubic spline coefficients for periodic end point conditions. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgicco  (x, yd, n, y, c, &s, wk);
```

Single precision:

```
ierr = ASL_rgicco  (x, yd, n, y, c, &s, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	y	{D∗ / R∗}	n	Output	0th order terms of cubic spline coefficients
5	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
6	s	{D∗ / R∗}	1	Output	Sum of the squares of the error ($\sum_{i=1}^{n}\{yd[i-1]-y[i-1]\}^2$)
7	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: n × (2 × n + 4)
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq 4$

(b) $x[0] < x[1] < \cdots < x[n-1]$ (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
4000	The minimum cross-validation value could not be found. (The data is uncorrelated.)	

(6) **Notes**

None

6.2.15 ASL_dgicct, ASL_rgicct
Cubic Spline Coefficients (Smoothing by Specifying a Control Variable)

(1) **Function**

ASL_dgicct or ASL_rgicct determines the smoothed (natural) cubic spline coefficients using the specified control variable.

(2) **Usage**

Double precision:

```
ierr = ASL_dgicct  (x, yd, w, n, sf, y, c, wk);
```

Single precision:

```
ierr = ASL_rgicct  (x, yd, w, n, sf, y, c, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	w	{D∗ / R∗}	n	Input	Relative weights at each sample points (w[i − 1] > 0) Usually 1.0 is set for non weighted one.
4	n	I	1	Input	Number of sample points
5	sf	{D / R}	1	Input	Control variable that controls degree of smoothing (sf > 0) The cubic spline function f(x) is determined so the following expression is satisfied. $\sum_{i=1}^{n}((f(x[i-1]) - yd[i-1])/w[i-1])^2 \leq sf$ where the equality holds if f(x) is not a linear function.
6	y	{D∗ / R∗}	n	Output	0th order terms of cubic spline coefficients
7	c	{D∗ / R∗}	3×(n − 1)	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (x[j − 1] ≤ t < x[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − x[j − 1]
8	wk	{D∗ / R∗}	7 × n + 14	Work	Work area.
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(b) x[0] < x[1] < ··· < x[n − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	

(6) **Notes**

(a) The input sf value is proportional to the sum of the squares of the distances between the input data points and the smoothed curve. For a large value, it is nearly a straight line; for a small value, it is nearly a complete interpolation.

(b) To have the spline coefficients become **a natural cubic spline**, let the second derivatives of the spline function be zero at x[0] and x[n − 1].

6.2.16 ASL_dgiccm, ASL_rgiccm
Cubic Spline Coefficients (Least Squares Method When Knot Positions are Set Automatically)

(1) **Function**

ASL_dgiccm or ASL_rgiccm automatically locates the optimum knot positions and obtains the best least squares approximation cubic spline coefficients. In addition optimum knot positions are obtained.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiccm  (x, yd, n, xk, nxk, &itmx, y, c, &s, iwk, wk1, wk2);
```

Single precision:

```
ierr = ASL_rgiccm  (x, yd, n, xk, nxk, &itmx, y, c, &s, iwk, wk1, wk2);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xk	{D∗ / R∗}	nxk	Input	Initial estimates of knot positions xk[i − 1] < xk[i], i = 1, ··· , nxk − 1 xk[0] ≤ x[0] xk[nxk − 1] ≥ x[n − 1]
				Output	Optimum knot positions
5	nxk	I	1	Input	Number of knots
6	itmx	I*	1	Input	Maximum number of iterations (15 is suitable)
				Output	Actual iteration count
7	y	{D∗ / R∗}	nxk	Output	0th order terms of cubic spline coefficients
8	c	{D∗ / R∗}	See Contents	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1] **Size**: 3×(nxk − 1)
9	s	{D∗ / R∗}	1	Output	Cubic spline approximation least squares error
10	iwk	I*	75	Work	Work area
11	wk1	{D∗ / R∗}	See Contents	Work	Work area **Size**: n × (nxk + 6)
12	wk2	{D∗ / R∗}	3811	Work	Work area
13	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq 2$, $2 \leq nxk \leq 28$, and $itmx \leq 20$

(b) $x[0] < x[1] < \cdots < x[n-1]$ (Ascending order)

(c) Sample points are distributed throughout the range determined by the endpoint knots.

(d) $xk[0] < xk[1] < \cdots < xk[nxk-1]$ (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1500	Calculations ended after itmx iterations without obtaining the minimum least squares error (s). (The data is nearly uncorrelated.)	Interpolation is performed using the current cubic spline coefficients and least squares error.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) was not satisfied.	
3030	Restriction (d) was not satisfied.	

(6) **Notes**

(a) Different cubic spline coefficients may be obtained and convergence status may be changed depending on the initial estimates of the knot positions.

(b) Usually values $y[0]$ and $y[nxk-1]$ become extrapolated ones using cubic spline coefficients.

6.2.17 ASL_dgiccn, ASL_rgiccn
Cubic Spline Coefficients (Least Squares Method When Knot Positions are Specified)

(1) **Function**

ASL_dgiccn or ASL_rgiccn obtains the least squares approximation cubic spline coefficients at the specified knots.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiccn  (x, yd, n, xk, &nxk, y, c, nkm, &s, isw, iwk, wk1, wk2);
```

Single precision:

```
ierr = ASL_rgiccn  (x, yd, n, xk, &nxk, y, c, nkm, &s, isw, iwk, wk1, wk2);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], yd[i − 1]), i = 1, ··· , n (x[i − 1] < x[i], i ≠ n)
2	yd	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values yd[i − 1]
3	n	I	1	Input	Number of sample points
4	xk	{D∗ / R∗}	See Contents	Input	Knot position abscissa values isw=0: Input all knots abscissa values for xk[i − 1] i = 1, ··· , nxk in the order: left endpoint knot, insertion knots, right endpoint knot. isw = 1 or 2: Input added knot abscissa values for xk[i − 1] i = 1, ··· , nxk isw=3: Input modified knot abscissa value for xk[0].
				Output	All knots abscissa values at that time (when isw = 4) **Size**: max(1, nxk)
5	nxk	I*	1	Input	Number of knots isw=0: Set number of all knots isw = 1 or 2: When you add knots, set number of added knots (positive value), when you delete knots, set sign changed number of deleted knots (negative value). (See Note (d)) isw=3: nxk=1
				Output	Number of all knots at that time (when isw = 4)
6	y	{D∗ / R∗}	nkm	Output	0th order terms of cubic spline coefficients

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
7	c	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	See Contents	Output	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1] **Size**: 3×(nkm − 1)
8	nkm	I	1	Input	Maximum number of knots
9	s	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	1	Output	Cubic spline approximation least squares error
10	isw	I	1	Input	Processing switch isw=0: Least squares approximation cubic spline coefficients are output for input knots (Initially, processing must be performed with isw=0). isw=1: When nxk ≥ 0, new cubic spline coefficients are calculated from the previous and added knots. if nxk < 0, cubic spline coefficients are recalculated deleting −nxk knots form the previously added knots. isw=2: Same as isw = 1. It is effective when you are again inserting knots after having deleted several knots. isw=3: The abscissa value of the knot input last is changed to the value of xk[0]. Only the least squares error is output; cubic spline coefficients are not calculated. isw=4: Number of knots at that time is returned to nxk and the abscissa values of each knot at that time are returned to xk[i − 1] i = 1, ⋯ , nxk.
11	iwk	I*	75	Work	Work area
12	wk1	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	See Contents	Work	Work area **Size**: n × (nkm + 6)
13	wk2	$\begin{Bmatrix}\text{D}*\\ \text{R}*\end{Bmatrix}$	3811	Work	Work area
14	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq 2, nxk \leq nkm \leq n, nkm \leq 28$, and if $isw = 0, nxk \geq 2$.

(b) $x[0] < x[1] < \cdots < x[n-1]$ (Ascending order)

(c) Knots must not be duplicated.

(d) Specify knots within the range from the left endpoint knot to the right endpoint knot. In addition, assign data within this range.

(e) When isw=4, then nxk $\geq$ (number of all knots at that time)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) was not satisfied.	
3020	Restriction (c) or (d) was not satisfied.	
3030	Restriction (e) was not satisfied.	

(6) **Notes**

(a) You must retain arrays x, yd, iwk, wk1, and wk2 until the optimum cubic spline coefficients are obtained using this function.

(b) Input the left endpoint knot to xk[0] and the right endpoint to xk[nxk − 1] with isw=0.

(c) Specify isw=0 to obtain the least squares approximation cubic spline coefficients for the first time. To modify knots thereafter, use isw= 1, 2 or 3. If you want to know the number of knots and knot positions, specify isw=4, and they are output in nxk and xk[i − 1] (ascending order), respectively.

(d) When deleting some knots with isw=1 or isw=2, knots are deleted in reversed order of knots added and descending order of knot position abscissa values. However endpoint knots are never deleted.
When nxk < −(Number of knots − 2) < 0, then all knots except endpoint knots are deleted. For example,

i. Using isw=0, knot position abscissa values {1.0, 2.0, 5.0, 7.0, 9.0} are input

ii. Using isw=1, knot position abscissa values {1.5, 2.5, 6.0} are added,

iii. Using isw=1, knot position abscissa values {3.0, 8.0} are added,

then current knot position abscissa values become to be {1.0, 1.5, 2.0, 2.5, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0}. At this time, if 3 knots are deleted by nxk = −3, then the knots of abscissa values {8.0, 3.0, 6.0} are deleted. If 6 knots are deleted by nxk = −6, then the knots of abscissa values {8.0, 3.0, 6.0, 2.5, 1.5, 7.0} are deleted. If you try to delete 9 knots by nxk = −9, 8 knots are deleted and 2 endpoint knots of abscissa values {1.0, 9.0} are remained.

(e) After modifying knot positions with isw=3, execute this function with isw= 1 or 2 and nxk=0 to obtain cubic spline coefficients at that time.

6.2.18 ASL_dgiscx, ASL_rgiscx
Interpolation Values According to Cubic Spline Coefficients

(1) **Function**

ASL_dgiscx or ASL_rgiscx calculates interpolation values of cubic spline function at specified points using given cubic spline coefficients.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiscx  (x, y, n, c, xl, yl, m);
```

Single precision:

```
ierr = ASL_rgiscx  (x, y, n, c, xl, yl, m);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Knots abscissa values x[i − 1] of cubic spline function (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	0th order terms of cubic spline coefficients
3	n	I	1	Input	Number of knots
4	c	{D∗ / R∗}	3×(n − 1)	Input	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1]
5	xl	{D∗ / R∗}	m	Input	Abscissa values of points where interpolation values are calculated
6	yl	{D∗ / R∗}	m	Output	Cubic spline interpolation values at xl[i − 1]
7	m	I	1	Input	Number of interpolation values
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2 and m > 0

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
3000	Restriction (a) was not satisfied.	Processing is aborted.

(6) **Notes**

None

6.2.19 ASL_dgidcy, ASL_rgidcy
Derivative Values According to Cubic Spline Coefficients

(1) **Function**

ASL_dgidcy or ASL_rgidcy calculates first and second derivatives of cubic spline function at specified points using given cubic spline coefficients.

(2) **Usage**

Double precision:

ierr = ASL_dgidcy (x, n, c, xl, dl, ddl, m);

Single precision:

ierr = ASL_rgidcy (x, n, c, xl, dl, ddl, m);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Knots abscissa values x[i − 1] of cubic spline function (x[i − 1] < x[i], i ≠ n)
2	n	I	1	Input	Number of knots
3	c	{D∗ / R∗}	3×(n − 1)	Input	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1] and y[j − 1] are 0th order terms of cubic spline coefficients.
4	xl	{D∗ / R∗}	m	Input	Abscissa values of points where derivatives of cubic spline function are calculated.
5	dl	{D∗ / R∗}	m	Output	First derivatives of cubic spline function at xl[i − 1] dl[i − 1] = (3.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)] where: xk[j − 1] ≤ xl[i − 1] < xk[j] d = xl[i − 1] − xk[j − 1]
6	ddl	{D∗ / R∗}	m	Output	Second derivatives of cubic spline function at xl[i − 1] ddl[i − 1] = 6.0 × c[2 + 3 × (j − 1)] × d + 2.0 × c[1 + 3 × (j − 1)] where: xk[j − 1] ≤ xl[i − 1] < xk[j] d = xl[i − 1] − xk[j − 1]
7	m	I	1	Input	Number of points where derivatives are calculated
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	xl[i − 1] was outside the interpolation interval range (xl[i − 1] < x[0] or xl[i − 1] > x[n − 1]).	The extrapolated value is output using the cubic spline coefficients at the endpoints.
3000	Restriction (a) was not satisfied.	Processing is aborted.

(6) **Notes**

None

6.2.20 ASL_dgiicz, ASL_rgiicz
Integral Value According to Cubic Spline Coefficients

(1) **Function**

ASL_dgiicz or ASL_rgiicz calculates the integral of the cubic spline function using given cubic spline coefficients.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiicz  (x, y, n, c, a, b, &q);
```

Single precision:

```
ierr = ASL_rgiicz  (x, y, n, c, a, b, &q);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Knots abscissa values x[i − 1] of cubic spline function (x[i − 1] < x[i], i ≠ n)
2	y	{D∗ / R∗}	n	Input	0th order terms of cubic spline coefficients
3	n	I	1	Input	Number of knots
4	c	{D∗ / R∗}	3×(n − 1)	Input	k-th order terms of cubic spline coefficients (k=1, 2, 3): c[(k − 1) + 3 × (j − 1)] The value of cubic spline function f(t) at abscissa value t (xk[j − 1] ≤ t < xk[j]): f(t) = ((c[2 + 3 × (j − 1)] × d + c[1 + 3 × (j − 1)]) × d + c[3 × (j − 1)]) × d + y[j − 1] where: d = t − xk[j − 1]
5	a	{D / R}	1	Input	Lower limit of the integration range
6	b	{D / R}	1	Input	Upper limit of the integration range
7	q	{D∗ / R∗}	1	Output	Integral value of cubic spline function over abscissa interval [a, b]
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 2

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	a or b was outside the interpolation interval	A function extrapolated from the cubic spline coefficients at the endpoints is integrated.
3000	Restriction (a) was not satisfied.	Processing is aborted.

(6) **Notes**

None

6.3 BICUBIC SPLINE (CURVED SURFACE INTERPOLATION)

6.3.1 ASL_dgisxb, ASL_rgisxb Interpolation Values

(1) **Function**

ASL_dgisxb or ASL_rgisxb performs a bicubic spline interpolation from data on a lattice and obtains interpolation values at new points on the lattice. Endpoint condition is "not-a-knot" and the knots are set equal to sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgisxb (x, nx, y, ny, z, xl, nxl, yl, nyl, fl, wk);

Single precision:

ierr = ASL_rgisxb (x, nx, y, ny, z, xl, nxl, yl, nyl, fl, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of sample points x[i − 1], i = 1, ··· , nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of sample points in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of sample points y[j − 1], j = 1, ··· , ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of sample points in Y direction
5	z	{D∗ / R∗}	nx×ny	Input	Array formed from function values z_{ij} at sample point (x[i − 1], y[j − 1]) z[(i − 1) + nx × (j − 1)] = z_{ij}
6	xl	{D∗ / R∗}	nxl	Input	X coordinate values of lattice points where interpolation values are calculated
7	nxl	I	1	Input	Number of interpolation points in X direction
8	yl	{D∗ / R∗}	nyl	Input	Y coordinate values of lattice points where interpolation values are calculated
9	nyl	I	1	Input	Number of interpolation points in Y direction
10	fl	{D∗ / R∗}	See Contents	Output	Array formed from bicubic spline function values f_{ij} at interpolation points (xl[i − 1], yl[j − 1]) fl[(i − 1) + nxl × (j − 1)] = f_{ij} Note that when ny > nyl, area fl[(i − 1) + nxl × (j − 1)] (i = 1, ··· , nxl; j = nyl + 1, ··· , ny) are used as a work area. After calculation the values corresponding to them become indefinite. **Size**: nxl×max(nyl, ny)
11	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: max{3 × (nx − 1), 3 × (ny − 1) + ny}.
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) nx ≥ 2, ny ≥ 2

(b) x[0] < x[1] < ··· < x[nx − 1] (Ascending order)

(c) y[0] < y[1] < ··· < y[ny − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	Interpolation point (xl[i−1], yl[j−1]) was out-side the interpolation region.	A value extrapolated by using the bicubic spline coefficients on the boundary is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) or (c) was not satisfied.	

(6) **Notes**

(a) Bicubic spline coefficients are not output.

(b) This function obtain bicubic spline interpolation value at all lattice point. To obtain the bicubic spline interpolation value at a specific point, it is more effective to call function 6.3.4 $\left\{\begin{matrix}\text{ASL\_dgicbp}\\ \text{ASL\_rgicbp}\end{matrix}\right\}$, and function 6.3.5 $\left\{\begin{matrix}\text{ASL\_dgisbx}\\ \text{ASL\_rgisbx}\end{matrix}\right\}$ successively.

(7) **Example**

(a) Problem
When the matrix Z formed from function values z_{ij} at sample point $(x_i, y_j) = (2 \times i - 1, 2 \times j - 1)$ ($i = 1, 2, 3$; $j = 1, 2, 3$) is given as follows:

$$Z = \begin{bmatrix} 2.0 & 0.5 & 1.0 \\ 4.0 & 1.0 & 2.0 \\ 3.0 & 0.75 & 1.5 \end{bmatrix}$$

perform bicubic spline interpolation using these points as sample points. In addition, obtain the values of the bicubic spline function at the lattice points $(xl_i, yl_j) = (0.5 \times (i+1), 0.5 \times (j+2))$ ($i = 1, 2$; $j = 1, 2$).

(b) Input data
$x[i-1] = x_i (i = 1, \cdots, nx)$,
$y[j-1] = y_j (j = 1, \cdots, ny)$,
$z[(i-1) + nx \times (j-1)] = z_{ij}$,
$xl[i-1] = xl_i (i = 1, \cdots, nxl)$,
$yl[j-1] = yl_j (j = 1, \cdots, nyl)$,
nx=3, ny=3, nxl=2 and nyl=2.

(c) Main program

```
/*      C interface example for ASL_dgisxb */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    int nx;
    double *y;
    int ny;
    double *f;
    double *xl;
    int nxl;
    double *yl;
    int nyl;
    double *fl;
```

```
double *wk;
int ierr;
int i,j,nwk;
FILE *fp;

fp = fopen( "dgisxb.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dgisxb ***\n" );
printf( "\n     ** Input **\n\n" );
nx=3;
ny=3;
nxl=2;
nyl=2;

f = ( double * )malloc((size_t)( sizeof(double) * (nx*ny) ));
if( f == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}

fl = ( double * )malloc((size_t)( sizeof(double) * (nxl*ny) ));
if( fl == NULL )
{
    printf( "no enough memory for array fl\n" );
    return -1;
}

nwk=( (3*nx-3>4*ny-3) ? 3*nx-3 : 4*ny-3 );
wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

xl = ( double * )malloc((size_t)( sizeof(double) * nxl ));
if( xl == NULL )
{
    printf( "no enough memory for array xl\n" );
    return -1;
}

yl = ( double * )malloc((size_t)( sizeof(double) * nyl ));
if( yl == NULL )
{
    printf( "no enough memory for array yl\n" );
    return -1;
}

printf( "\tn   = %6d\n", nx  );
printf( "\tn   = %6d\n", ny  );
printf( "\tnxl = %6d\n", nxl );
printf( "\tnyl = %6d\n", nyl );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &y[i] );
}
for( j=0 ; j<ny ; j++ )
{
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &f[i+nx*j] );
    }
}
for( i=0 ; i<nxl ; i++ )
{
    fscanf( fp, "%lf", &xl[i] );
```

```
    }
    for( i=0 ; i<nyl ; i++ )
    {
        fscanf( fp, "%lf", &yl[i] );
    }

    printf( "\n" );
    printf( "\tCoordinates (x,y)\n\n" );
    printf( "\t     i     x[i]     y[i]\n");
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t%6d %8.3g %8.3g\n", i,x[i],y[i] );
    }

    printf( "\n" );
    printf( "\tSpecified Points\n\n" );
    printf( "\t     j    xl[j]    yl[j]\n");
    for( j=0 ; j<nxl ; j++ )
    {
        printf( "\t%6d %8.3g %8.3g\n", j,xl[j],yl[j] );
    }

    printf( "\n" );
    printf( "\tFunction Values\n\n" );
    printf( "\t  z[i,j]\n");
    printf( "\t                j=0      j=1      j=2\n" );
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t i=%6d",i );
        for( j=0 ; j<ny ; j++ )
        {
            printf( " %8.3g", f[i+nx*j] );
        }
        printf( "\n" );
    }

    fclose( fp );

    ierr = ASL_dgisxb(x, nx, y, ny, f, xl, nxl, yl, nyl, fl, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\tEstimated Function Values in (x,y)\n\n" );
    printf( "\t  fl[i,j]\n" );
    printf( "\t                j=0      j=1\n" );
    for( i=0 ; i<nxl ; i++ )
    {
        printf( "\t i=%6d",i );
        for( j=0 ; j<nyl ; j++ )
        {
            printf( " %8.3g", fl[i+nxl*j] );
        }
        printf( "\n" );
    }

    free( x );
    free( y );
    free( f );
    free( xl );
    free( yl );
    free( fl );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgisxb ***

 ** Input **

n   =      3
n   =      3
nxl =      2
nyl =      2

Coordinates (x,y)

     i     x[i]     y[i]
     0        1        1
     1        3        3
     2        5        5

Specified Points
```

```
     j    xl[j]    yl[j]
     0        1      1.5
     1      1.5        2

Function Values

  z[i,j]
               j=0      j=1      j=2
 i=     0        2      0.5        1
 i=     1        4        1        2
 i=     2        3     0.75      1.5

 ** Output **

ierr =      0

Estimated Function Values in (x,y)

  fl[i,j]
               j=0      j=1
 i=     0     1.44        1
 i=     1        2     1.39
```

6.3.2 ASL_dgidyb, ASL_rgidyb
Mixed Partial Derivative Values and Bicubic Spline Coefficients

(1) **Function**

ASL_dgidyb or ASL_rgidyb obtains bicubic spline coefficients from lattice data and calculates mixed partial derivatives of the bicubic spline function at arbitrary points. Endpoint condition is "not-a-knot" and the knots are set equal to sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgidyb  (x, nx, y, ny, z, xl, yl, dl, c, wk);
```

Single precision:

```
ierr = ASL_rgidyb  (x, nx, y, ny, z, xl, yl, dl, c, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of sample points x[i − 1], i = 1, ··· , nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of sample points in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of sample points y[j − 1], j = 1, ··· , ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of sample points in Y direction
5	z	{D∗ / R∗}	nx×ny	Input	Array formed from function values z_{ij} at sample point (x[i − 1], y[j − 1]) z[(i − 1) + nx × (j − 1)] = z_{ij}
6	xl	{D / R}	1	Input	X coordinate values of lattice points where mixed partial derivatives are calculated
7	yl	{D / R}	1	Input	Y coordinate values of lattice points where mixed partial derivatives are calculated
8	dl	{D∗ / R∗}	6	Output	Obtained value of bicubic spline function $f(x, y)$ at point (x, y)=(xl, yl) and its mixed partial derivative dl[0]=$f(x, y)$, dl[1]=$\partial f/\partial x$, dl[2]=$\partial f/\partial y$, dl[3]=$\partial^2 f/(\partial x \partial y)$ dl[4]=$\partial^2 f/(\partial x)^2$ dl[5]=$\partial^2 f/(\partial y)^2$
9	c	{D∗ / R∗}	4×nx×ny	Output	Spline coefficients (See Note (a))
10	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: 2 × nx × ny + 2 × max(nx, ny).
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) nx ≥ 4, ny ≥ 4

(b) x[0] < x[1] < ··· < x[nx − 1] (Ascending order)

(c) y[0] < y[1] < ··· < y[ny − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The point where the partial derivative was to be calculated was outside the interpolation region.	A value extrapolated by using the bicubic spline coefficients on the boundary is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) or (c) was not satisfied.	

(6) **Notes**

(a) This function obtain the values z(i, j), $z_x(i,j)$, $z_y(i,j)$, $z_{xy}(i,j)$ as bicubic spline coefficients instead of the values $\alpha_{e,r}^{i,j}$ (See Section 6.1.2).

(b) To continue further obtaining interpolation values, partial derivatives or double integrals, you have called this function and then call function 6.3.5 $\begin{Bmatrix}\text{ASL\_dgisbx}\\ \text{ASL\_rgisbx}\end{Bmatrix}$, 6.3.6 $\begin{Bmatrix}\text{ASL\_dgidby}\\ \text{ASL\_rgidby}\end{Bmatrix}$ or 6.3.7 $\begin{Bmatrix}\text{ASL\_dgiibz}\\ \text{ASL\_rgiibz}\end{Bmatrix}$, respectively. In this case, contents of array x, y, c and variable nx and ny must input to corresponding arguments of the succeeding function. This enables you to eliminate unnecessary calculations since you calculate bicubic spline coefficients only once.

(7) **Example**

(a) Problem
When the matrix Z formed from function values z_{ij} at sample point $(x_i, y_j) = (2 \times i - 1, 2 \times j - 1)(i = 1, 2, 3, 4;\ j = 1, 2, 3, 4)$ is given as follows:

$$Z = \begin{bmatrix} 2.0 & 0.5 & 1.0 & 3.5 \\ 4.0 & 1.0 & 2.0 & 7.0 \\ 3.0 & 0.75 & 1.5 & 5.25 \\ -1.0 & -0.25 & -0.5 & -1.75 \end{bmatrix}$$

perform bicubic spline interpolation using these points as sample points. In addition, obtain the partial derivatives of the bicubic spline function at the specified points.

(b) Input data
$x[i-1] = x_i(i = 1, \cdots, nx)$, $y[j-1] = y_j(j = 1, \cdots, ny)$, $z[(i-1) + nx \times (j-1)] = z_{ij}$, nx=4, ny=4, xl=1.5 and yl=1.5.

(c) Main program

```
/*      C interface example for ASL_dgidyb */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    int nx;
    double *y;
    int ny;
    int nwk;
    double *f;
    double xl;
    double yl;
    double ds[6];
    double *c;
    double *wk;
    int ierr;
```

```
int i,j,k,l;
FILE *fp;

fp = fopen( "dgidyb.dat", "r" );
if( fp == NULL )
{
    printf( "file open error\n" );
    return -1;
}

printf( "    *** ASL_dgidyb ***\n" );
printf( "\n     ** Input **\n\n" );
nx=4;
ny=4;
xl=1.5;
yl=1.5;

c = ( double * )malloc((size_t)( sizeof(double) * (2*nx*2*ny) ));
if( c == NULL )
{
    printf( "no enough memory for array c\n" );
    return -1;
}

f = ( double * )malloc((size_t)( sizeof(double) * (nx*ny) ));
if( f == NULL )
{
    printf( "no enough memory for array f\n" );
    return -1;
}

nwk = 2 * (nx * ny + ( (nx>ny)?nx:ny ));
wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * ny ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}


printf( "\tnx  = %6d\n", nx  );
printf( "\tny  = %6d\n", ny  );
printf( "\n\tSpecified Points   xl = %8.3g\n", xl );
printf( "\tSpecified Points   yl = %8.3g\n", yl );
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &x[i] );
}
for( i=0 ; i<nx ; i++ )
{
    fscanf( fp, "%lf", &y[i] );
}
for( j=0 ; j<ny ; j++ )
{
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &f[i+nx*j] );
    }
}

printf( "\n\tCoordinates (x,y)\n\n" );
printf( "\t     i                x[i]               y[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],y[i] );
}

printf( "\n" );
printf( "\tFunction Values\n\n" );
printf( "\t  z[i,j]\n");
printf( "\t\t            j=0          j=1          j=2          j=3\n" );
for( i=0 ; i<nx ; i++ )
{
    printf( "\t  i=%6d",i );
    for( j=0 ; j<ny ; j++ )
```

```
        {
            printf( "     %8.3g", f[i+nx*j] );
        }
        printf( "\n" );
    }

    fclose( fp );

    ierr = ASL_dgidyb(x, nx, y, ny, f, xl, yl, ds, c, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\tThe Partial Derivatives\n\n" );
    for( i=0 ; i<6 ; i++ )
    {
        printf( "\t  dl[%6d]=%8.3g\n", i,ds[i] );
    }

    printf( "\n" );
    printf( "\tSpline Coefficients\n" );
    printf( "\n" );
    for( l=0 ; l<4 ; l++ )
    {
        for( k=0 ; k<2 ; k++ )
        {
            for( j=0 ; j<4 ; j++ )
            {
    printf( "\t  c[     0,%6d,%6d,%6d]=%8.3g  ",j,k,l,c[  2*(j+nx*(k+2*l))] );
    printf( "  c[     1,%6d,%6d,%6d]=%8.3g\n",j,k,l,c[1+2*(j+nx*(k+2*l))] );
            }
        }
        printf( "\n" );
    }

    free( x );
    free( y );
    free( f );
    free( c );
    free( wk );


    return 0;
}
```

(d) Output results

```
*** ASL_dgidyb ***

 ** Input **

nx  =      4
ny  =      4

Specified Points   xl =      1.5
Specified Points   yl =      1.5

Coordinates (x,y)

     i                x[i]               y[i]
     0                  1                  1
     1                  3                  3
     2                  5                  5
     3                  7                  7

Function Values

  z[i,j]
                 j=0          j=1          j=2          j=3
  i=     0           2          0.5            1          3.5
  i=     1           4            1            2            7
  i=     2           3         0.75          1.5         5.25
  i=     3          -1        -0.25         -0.5        -1.75

 ** Output **

ierr =      0

The Partial Derivatives

  dl[     0]=        2
  dl[     1]=    0.988
  dl[     2]=    -1.39
  dl[     3]=   -0.688
  dl[     4]=   -0.539
  dl[     5]=    0.695

Spline Coefficients

  c[     0,     0,     0,     0]=        2    c[     1,     0,     0,     0]=     1.75
```

```
c[  0,  1,  0,  0]=      4    c[  1,  1,  0,  0]=   0.25
c[  0,  2,  0,  0]=      3    c[  1,  2,  0,  0]=  -1.25
c[  0,  3,  0,  0]=     -1    c[  1,  3,  0,  0]=  -2.75
c[  0,  0,  1,  0]=  -1.25    c[  1,  0,  1,  0]=  -1.09
c[  0,  1,  1,  0]=   -2.5    c[  1,  1,  1,  0]= -0.156
c[  0,  2,  1,  0]=  -1.88    c[  1,  2,  1,  0]=  0.781
c[  0,  3,  1,  0]=  0.625    c[  1,  3,  1,  0]=   1.72

c[  0,  0,  0,  1]=    0.5    c[  1,  0,  0,  1]=  0.438
c[  0,  1,  0,  1]=      1    c[  1,  1,  0,  1]= 0.0625
c[  0,  2,  0,  1]=   0.75    c[  1,  2,  0,  1]= -0.313
c[  0,  3,  0,  1]=  -0.25    c[  1,  3,  0,  1]= -0.688
c[  0,  0,  1,  1]=  -0.25    c[  1,  0,  1,  1]= -0.219
c[  0,  1,  1,  1]=   -0.5    c[  1,  1,  1,  1]=-0.0313
c[  0,  2,  1,  1]= -0.375    c[  1,  2,  1,  1]=  0.156
c[  0,  3,  1,  1]=  0.125    c[  1,  3,  1,  1]=  0.344

c[  0,  0,  0,  2]=      1    c[  1,  0,  0,  2]=  0.875
c[  0,  1,  0,  2]=      2    c[  1,  1,  0,  2]=  0.125
c[  0,  2,  0,  2]=    1.5    c[  1,  2,  0,  2]= -0.625
c[  0,  3,  0,  2]=   -0.5    c[  1,  3,  0,  2]=  -1.38
c[  0,  0,  1,  2]=   0.75    c[  1,  0,  1,  2]=  0.656
c[  0,  1,  1,  2]=    1.5    c[  1,  1,  1,  2]= 0.0938
c[  0,  2,  1,  2]=   1.13    c[  1,  2,  1,  2]= -0.469
c[  0,  3,  1,  2]= -0.375    c[  1,  3,  1,  2]=  -1.03

c[  0,  0,  0,  3]=    3.5    c[  1,  0,  0,  3]=  3.06
c[  0,  1,  0,  3]=      7    c[  1,  1,  0,  3]=  0.438
c[  0,  2,  0,  3]=   5.25    c[  1,  2,  0,  3]=  -2.19
c[  0,  3,  0,  3]=  -1.75    c[  1,  3,  0,  3]=  -4.81
c[  0,  0,  1,  3]=   1.75    c[  1,  0,  1,  3]=   1.53
c[  0,  1,  1,  3]=    3.5    c[  1,  1,  1,  3]=  0.219
c[  0,  2,  1,  3]=   2.63    c[  1,  2,  1,  3]=  -1.09
c[  0,  3,  1,  3]= -0.875    c[  1,  3,  1,  3]=  -2.41
```

6.3.3 ASL_dgiizb, ASL_rgiizb Double Integral Value

(1) **Function**

ASL_dgiizb or ASL_rgiizb calculates double integral of the bicubic spline function obtained from data on lattice. The knots are set equal to sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiizb  (x, nx, y, ny, z, ax, bx, cy, dy, &q, wk);
```

Single precision:

```
ierr = ASL_rgiizb  (x, nx, y, ny, z, ax, bx, cy, dy, &q, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of sample points x[i − 1], i = 1, ··· , nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of sample points in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of sample points y[j − 1], j = 1, ··· , ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of sample points in Y direction
5	z	{D∗ / R∗}	nx×ny	Input	Array formed from function values z_{ij} at sample point (x[i − 1], y[j − 1]) z[(i − 1) + nx × (j − 1)] = z_{ij}
6	ax	{D / R}	1	Input	Lower limit of the integration region in the X direction
7	bx	{D / R}	1	Input	Upper limit of the integration region in the X direction
8	cy	{D / R}	1	Input	Lower limit of the integration region in the Y direction
9	dy	{D / R}	1	Input	Upper limit of the integration region in the Y direction

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	q	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	1	Output	Value of the double integral over the rectangular region $[\text{ax}, \text{bx}] \times [\text{cy}, \text{dy}]$
11	wk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $(\text{ny}+5) \times \text{nx} + \text{ny} - 1 + \max(5 \times \text{nx} - 4, 5 \times \text{ny} - 2)$
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{nx} \geq 2, \text{ny} \geq 2$

(b) $\text{x}[0] < \text{x}[1] < \cdots < \text{x}[\text{nx}-1]$ (Ascending order)

(c) $\text{y}[0] < \text{y}[1] < \cdots < \text{y}[\text{ny}-1]$ (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The rectangular region $[\text{ax}, \text{bx}] \times [\text{cy}, \text{dy}]$ went into outside the interpolation region.	A value extrapolated by using the bicubic spline coefficients on the boundary is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) or (c) was not satisfied.	

(6) **Notes**

(a) Bicubic spline coefficients are not calculated. (See Section 6.1.2)

(b) The calculation method differs from that used by function 6.3.7 $\left\{\begin{matrix}\text{ASL\_dgiibz}\\ \text{ASL\_rgiibz}\end{matrix}\right\}$. Therefore, the results may not match completely.

(7) **Example**

(a) Problem

When the matrix Z formed from function values z_{ij} at sample points $(\text{x}_\text{i}, \text{y}_\text{j}) = (\text{i}-1, \text{j}-1)$, $(\text{i} = 1, 2, 3, 4;\ \text{j} = 1, 2, 3, 4)$ is given as follows:

$$Z = \begin{bmatrix} 8.0 & 7.0 & 6.0 & 5.0 \\ 2.0 & 3.0 & 4.0 & 5.0 \\ -4.0 & -1.0 & 2.0 & 5.0 \\ -10.0 & -5.0 & 0.0 & 5.0 \end{bmatrix}$$

perform bicubic spline interpolation using these points as sample points. In addition, obtain the double integral of the bicubic spline function over a rectangular region $[0.5, 2.5] \times [0.5, 2.5]$.

(b) Input data

$\text{x}[\text{i}-1] = \text{x}_\text{i} (\text{i} = 1, \cdots, \text{nx})$,
$\text{y}[\text{j}-1] = \text{y}_\text{j} (\text{j} = 1, \cdots, \text{ny})$,
$\text{z}[(\text{i}-1) + \text{nx} \times (\text{j}-1)] = \text{z}_{\text{ij}}$,
nx=4, ny=4, ax=0.5, bx=2.5, cy=0.5 and dy=2.5.

(c) Main program

```
/*      C interface example for ASL_dgiizb */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    int nx;
    double *y;
    int ny;
    double *f;
    double ax;
    double bx;
    double cy;
    double dy;
    double  q;
    double *wk;
    int ierr;
    int i,j,nwk;
    FILE *fp;

    fp = fopen( "dgiizb.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgiizb ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=4;
    ny=4;
    ax=0.5;
    bx=2.5;
    cy=0.5;
    dy=2.5;

    f = ( double * )malloc((size_t)( sizeof(double) * (nx*ny) ));
    if( f == NULL )
    {
        printf( "no enough memory for array f\n" );
        return -1;
    }

    nwk=(ny+5)*nx+ny-1+( (5*nx-4>5*ny-2) ? 5*nx-4 : 5*ny-2 );
    wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }

    y = ( double * )malloc((size_t)( sizeof(double) * ny ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    printf( "\tnx  = %6d\n", nx  );
    printf( "\tny  = %6d\n", ny  );
    printf( "\n\tLimits of Integration   ax=%8.3g\n", ax );
    printf( "\t                        bx=%8.3g\n", bx );
    printf( "\t                        cy=%8.3g\n", cy );
    printf( "\t                        dy=%8.3g\n", dy );
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
    }
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
    }
    for( j=0 ; j<ny ; j++ )
    {
        for( i=0 ; i<nx ; i++ )
        {
            fscanf( fp, "%lf", &f[i+nx*j] );
        }
    }
```

```
    printf( "\n" );
    printf( "\tCoordinates (x,y)\n\n" );
    printf( "\t     i      x[i]    y[i]\n");
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t%6d %8.3g %8.3g\n", i,x[i],y[i] );
    }

    printf( "\n" );
    printf( "\tFunction Values\n\n" );
    printf( "\t  z[i,j]\n");
    printf( "\t                j=0      j=1      j=2      j=3\n" );
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t i=%6d ",i );
        for( j=0 ; j<ny ; j++ )
        {
            printf( " %8.3g", f[i+nx*j] );
        }
        printf( "\n" );
    }

    fclose( fp );

    ierr = ASL_dgiizb(x, nx, y, ny, f, ax, bx, cy, dy, &q, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\tDouble Integral\n\n" );
    printf( "\t  q=%8.3g\n",q );

    free( x );
    free( y );
    free( f );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgiizb ***

 ** Input **

nx  =      4
ny  =      4

Limits of Integration   ax=     0.5
                        bx=     2.5
                        cy=     0.5
                        dy=     2.5

Coordinates (x,y)

     i      x[i]    y[i]
     0        0       0
     1        1       1
     2        2       2
     3        3       3

Function Values

  z[i,j]
                j=0      j=1      j=2      j=3
 i=     0         8        7        6        5
 i=     1         2        3        4        5
 i=     2        -4       -1        2        5
 i=     3       -10       -5        0        5

 ** Output **

ierr =      0

Double Integral

  q=       8
```

6.3.4 ASL_dgicbp, ASL_rgicbp Bicubic Spline Coefficients

(1) **Function**

ASL_dgicbp or ASL_rgicbp obtains bicubic spline coefficients for "not-a-knot" endpoint conditions. The knots are set equal to sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgicbp (x, nx, y, ny, z, c, wk);

Single precision:

ierr = ASL_rgicbp (x, nx, y, ny, z, c, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of sample points x[i − 1], i = 1, ··· , nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of sample points in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of sample points y[j − 1], j = 1, ··· , ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of sample points in Y direction
5	z	{D∗ / R∗}	nx×ny	Input	Array formed from function values z_{ij} at sample point (x[i − 1], y[j − 1]) z[(i − 1) + nx × (j − 1)] = z_{ij}
6	c	{D∗ / R∗}	4×nx×ny	Output	Bicubic spline coefficients (See Notes (a))
7	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: 2 × nx × ny + 2 × max(nx, ny)
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) nx ≥ 4, ny ≥ 4

(b) x[0] < x[1] < ··· < x[nx − 1] (Ascending order)

(c) y[0] < y[1] < ··· < y[ny − 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3010	Restriction (b) or (c) was not satisfied.	

(6) **Notes**

(a) This function obtain the values z(i, j), z_x(i, j), z_y(i, j), z_{xy}(i, j) as bicubic spline coefficients instead of the values $\alpha_{e,r}^{i,j}$ (See Section 6.1.2).

6.3.5 ASL_dgisbx, ASL_rgisbx
Interpolation Values According to Bicubic Spline Coefficients

(1) **Function**

ASL_dgisbx or ASL_rgisbx calculates an interpolation value of bicubic spline function using given bicubic spline coefficients.

(2) **Usage**

Double precision:

```
ierr = ASL_dgisbx  (x, nx, y, ny, c, xl, yl, &fl);
```

Single precision:

```
ierr = ASL_rgisbx  (x, nx, y, ny, c, xl, yl, &fl);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of knots x[i − 1], i = $1, \cdots,$ nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of knots in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of knots y[j − 1], j = $1, \cdots,$ ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of knots in Y direction
5	c	{D∗ / R∗}	4×nx×ny	Input	Bicubic spline coefficients (See Notes (a))
6	xl	{D / R}	1	Input	X coordinate value of point where interpolation value is calculated.
7	yl	{D / R}	1	Input	Y coordinate value of point where interpolation value is calculated.
8	fl	{D∗ / R∗}	1	Output	Bicubic spline function value at point (xl, yl)
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) nx ≥ 4, ny ≥ 4

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The interpolation point (xl, yl) was outside the interpolation region.	A value extrapolated by using the bicubic spline coefficients on the boundary is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) The bicubic spline coefficients for this function input are z(i, j), z_x(i, j), z_y(i, j), z_{xy}(i, j), not $\alpha_{e,r}^{i,j}$ (See Section 6.1.2).

(b) This function obtain bicubic spline interpolation value at a specific point. To obtain the bicubic spline interpolation value at all lattice point, it is more effective to call function 6.3.1 $\left\{\begin{matrix}\text{ASL\_dgisxb}\\ \text{ASL\_rgisxb}\end{matrix}\right\}$.

6.3.6 ASL_dgidby, ASL_rgidby
Mixed Partial Derivative Values According to Bicubic Spline Coefficients

(1) **Function**

ASL_dgidby or ASL_rgidby calculates mixed partial derivatives of bicubic spline function using given bicubic spline coefficients.

(2) **Usage**

Double precision:

ierr = ASL_dgidby (x, nx, y, ny, c, xl, yl, dl);

Single precision:

ierr = ASL_rgidby (x, nx, y, ny, c, xl, yl, dl);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of knots x[i − 1], i = $1, \cdots,$ nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of knots in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of knots y[j − 1], j = $1, \cdots,$ ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of knots in Y direction
5	c	{D∗ / R∗}	4×nx×ny	Input	Bicubic spline coefficients (See Notes (a))
6	xl	{D / R}	1	Input	X coordinate value of point where mixed partial derivative is calculated.
7	yl	{D / R}	1	Input	Y coordinate value of point where mixed partial derivative is calculated.
8	dl	{D∗ / R∗}	6	Output	Bicubic spline function value $f(x, y)$ and its mixed partial derivatives at point (x, y)=(xl, yl): dl[0]=f, dl[1]=$\partial f/\partial x$, dl[2]=$\partial f/\partial y$, dl[3]=$\partial^2 f/(\partial x \partial y)$ dl[4]=$\partial^2 f/(\partial x)^2$ dl[5]=$\partial^2 f/(\partial y)^2$
9	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{nx} \geq 4, \mathrm{ny} \geq 4$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The point (xl, yl) was outside the interpolation region.	A value extrapolated by using the bicubic spline coefficients on the boundary is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) The bicubic spline coefficients for this function input are z(i, j), $z_x(i, j)$, $z_y(i, j)$, $z_{xy}(i, j)$, not $\alpha_{e,r}^{i,j}$ (See Section 6.1.2).

6.3.7 ASL_dgiibz, ASL_rgiibz
Double Integral Value According to Bicubic Spline Coefficients

(1) **Function**

ASL_dgiibz or ASL_rgiibz obtains the value of an integral of bicubic spline function over a rectangular region using given bicubic spline coefficients.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiibz  (x, nx, y, ny, c, ax, bx, cy, dy, &q);
```

Single precision:

```
ierr = ASL_rgiibz  (x, nx, y, ny, c, ax, bx, cy, dy, &q);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	nx	Input	X coordinate values of knots x[i − 1], i = $1, \cdots,$ nx (x[i − 1] < x[i], i ≠ nx)
2	nx	I	1	Input	Number of knots in X direction
3	y	{D∗ / R∗}	ny	Input	Y coordinate values of knots y[j − 1], j = $1, \cdots,$ ny (y[j − 1] < y[j], j ≠ ny)
4	ny	I	1	Input	Number of knots in Y direction
5	c	{D∗ / R∗}	4×nx×ny	Input	Bicubic spline coefficients (See Notes (a))
6	ax	{D / R}	1	Input	Lower limit of the integration region in the X direction
7	bx	{D / R}	1	Input	Upper limit of the integration region in the X direction
8	cy	{D / R}	1	Input	Lower limit of the integration region in the Y direction
9	dy	{D / R}	1	Input	Upper limit of the integration region in the Y direction
10	q	{D∗ / R∗}	1	Output	Value of the double integral over the rectangular region [ax, bx] × [cy, dy]
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{nx} \geq 4, \mathrm{ny} \geq 4$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	The rectangular region $[\mathrm{ax}, \mathrm{bx}] \times [\mathrm{cy}, \mathrm{dy}]$ went into outside the interpolation region.	A value extrapolated by using the bicubic spline coefficients on the boundary is output.
3000	Restriction (a) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) The bicubic spline coefficients for this function input are z(i, j), $\mathrm{z_x}$(i, j), $\mathrm{z_y}$(i, j), $\mathrm{z_{xy}}$(i, j), not $\alpha_{e,r}^{i,j}$ (See Section 6.1.2).

(b) The calculation method differs from that used by function 6.3.3 $\left\{\begin{matrix}\text{ASL\_dgiizb}\\ \text{ASL\_rgiizb}\end{matrix}\right\}$. Therefore, the results may not match completely.

6.4 PLANE DATA INTERPOLATION

6.4.1 ASL_dgispo, ASL_rgispo
Open Curve Interpolation

(1) **Function**

ASL_dgispo or ASL_rgispo performs a plane data interpolation when the string of input data points takes the shape of an open curve. The knots are set equal to sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgispo  (x, y, n, is, ie, m, xo, yo, wk);
```

Single precision:

```
ierr = ASL_rgispo  (x, y, n, is, ie, m, xo, yo, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]), i = 1, ··· , n (Input them in the order they are to be interpolated)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
3	n	I	1	Input	Number of sample points
4	is	I	1	Input	Sample point element number of interpolation starting point Interpolation starting point is (x[is − 1], y[is − 1])
5	ie	I	1	Input	Sample point element number of interpolation end point Interpolation end point is (x[ie − 1], y[ie − 1])
6	m	I	1	Input	Number of interpolation data values (=Number of subdivisions+1)
7	xo	{D∗ / R∗}	m	Output	Abscissa value xo[i − 1] of interpolated point
8	yo	{D∗ / R∗}	m	Output	Ordinate value yo[i − 1] of interpolated point

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
9	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area **Size**: $4 \times \text{n} + \text{m} - 3$
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{n} \geq 2$

(b) $\text{m} \geq 2$

(c) $1 \leq \text{is} \leq \text{n}$

(d) $1 \leq \text{ie} \leq \text{n}$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	X coordinate became to be out of interpolation interval.	The extrapolated value is output for points out of interpolation interval.
3000	Restriction (a), (b), (c) or (d) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) Generally, obtained interpolation points are not equally spaced.

(7) **Example**

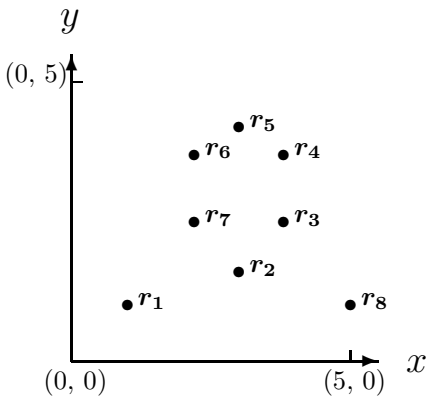
(a) Problem

If data points are given as shown in the figure, then interpolate points in the following order with 20 points.

Order: $\boldsymbol{r_1} \to \boldsymbol{r_2} \to \boldsymbol{r_3} \to \boldsymbol{r_4} \to \boldsymbol{r_5} \to \boldsymbol{r_6} \to \boldsymbol{r_7} \to \boldsymbol{r_2} \to \boldsymbol{r_8}$

where:

$\boldsymbol{r_1} = (1.0, 1.0)$, $\boldsymbol{r_2} = (3.0, 1.6)$, $\boldsymbol{r_3} = (3.8, 2.5)$, $\boldsymbol{r_4} = (3.8, 3.7)$,
$\boldsymbol{r_5} = (3.0, 4.2)$, $\boldsymbol{r_6} = (2.2, 3.7)$, $\boldsymbol{r_7} = (2.2, 2.5)$, $\boldsymbol{r_8} = (5.0, 1.0)$.



(b) Input data

Array x and y, n = 9, is = 1, ie = 9 and m = 20.

(c) Main program

```
/*      C interface example for ASL_dgispo */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int nx;
    int is;
    int ie;
    int m;
    double *xo;
    double *yo;
    double *wk;
    int ierr;
    int i,nwk;
    FILE *fp;

    fp = fopen( "dgispo.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgispo ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=9;
    m=20;
    is=1;
    ie=9;

    nwk=4*nx+m-3;
    wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }

    y = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    xo = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( xo == NULL )
    {
        printf( "no enough memory for array xo\n" );
        return -1;
    }

    yo = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( yo == NULL )
    {
        printf( "no enough memory for array yo\n" );
        return -1;
    }

    printf( "\tn   = %6d\n", nx  );
    printf( "\tis  = %6d\n", is  );
    printf( "\tie  = %6d\n", ie  );
    printf( "\tm   = %6d\n", m   );
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
    }
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
    }

    printf( "\n" );
```

```
    printf( "\tCoordinates (x,y)\n\n" );
    printf( "\t     i               x[i]               y[i]\n");
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],y[i] );
    }

    fclose( fp );

    ierr = ASL_dgispo(x, y, nx, is, ie, m, xo, yo, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\t     i               xo[i]              yo[i]\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf( "\t%6d           %8.3g           %8.3g\n", i,xo[i],yo[i] );
    }

    free( x );
    free( y );
    free( xo );
    free( yo );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgispo ***

 ** Input **

n   =      9
is  =      1
ie  =      9
m   =     20

Coordinates (x,y)

     i                x[i]               y[i]
     0                   1                  1
     1                   3                1.6
     2                 3.8                2.5
     3                 3.8                3.7
     4                   3                4.2
     5                 2.2                3.7
     6                 2.2                2.5
     7                   3                1.6
     8                   5                  1

 ** Output **

ierr =      0

     i               xo[i]              yo[i]
     0                   1                  1
     1                1.54               1.06
     2                2.11                1.2
     3                2.66               1.41
     4                3.17               1.72
     5                3.58               2.12
     6                3.85               2.64
     7                3.94               3.22
     8                3.75               3.77
     9                3.29               4.14
    10                2.71               4.14
    11                2.25               3.77
    12                2.06               3.22
    13                2.15               2.64
    14                2.42               2.12
    15                2.83               1.72
    16                3.34               1.41
    17                3.89                1.2
    18                4.46               1.06
    19                   5                  1
```

6.4.2 ASL_dgispr, ASL_rgispr Closed Curve Interpolation

(1) **Function**

ASL_dgispr or ASL_rgispr performs a plane data interpolation when the string of input data points takes the shape of a closed curve. The knots are set equal to sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgispr (x, y, n, is, ie, m, xo, yo, wk);

Single precision:

ierr = ASL_rgispr (x, y, n, is, ie, m, xo, yo, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]), i = 1, ··· , n (Input them in the order they are to be interpolated)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
3	n	I	1	Input	Number of sample points
4	is	I	1	Input	Sample point element number of interpolation starting point Interpolation starting point is (x[is − 1], y[is − 1])
5	ie	I	1	Input	Sample point element number of interpolation end point Interpolation end point is (x[ie − 1], y[ie − 1])
6	m	I	1	Input	Number of interpolation data values (=Number of subdivisions+1)
7	xo	{D∗ / R∗}	m	Output	Abscissa value xo[i − 1] of interpolated point
8	yo	{D∗ / R∗}	m	Output	Ordinate value yo[i − 1] of interpolated point
9	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: 9 × n + m − 3
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq 4$

(b) $m \geq 2$

(c) $1 \leq is \leq n$

(d) $1 \leq ie \leq n$

(e) $x[0] = x[n-1]$, $y[0] = y[n-1]$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	X coordinate became to be out of interpolation interval.	The extrapolated value is output for points out of interpolation interval.
2000	Restriction (e) was not satisfied.	Processing is performed regarding $(x[n-1], y[n-1])$ as (x[0], y[0]).
3000	Restriction (a), (b), (c) or (d) was not satisfied.	Processing is aborted.

(6) **Notes**

(a) Generally, obtained interpolation points are not equally spaced.

(7) **Example**

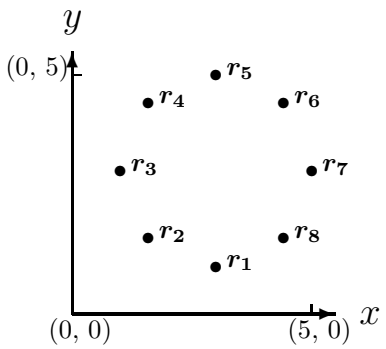
(a) Problem

If data points are given as shown in the figure, then interpolate points in the following order with 20 points.

Order: $\boldsymbol{r_1} \rightarrow \boldsymbol{r_2} \rightarrow \boldsymbol{r_3} \rightarrow \boldsymbol{r_4} \rightarrow \boldsymbol{r_5} \rightarrow \boldsymbol{r_6} \rightarrow \boldsymbol{r_7} \rightarrow \boldsymbol{r_8} \rightarrow \boldsymbol{r_1}$

where:

$\boldsymbol{r_1} = (3.0, 1.0)$, $\boldsymbol{r_2} = (1.5858, 1.5858)$, $\boldsymbol{r_3} = (1.0, 3.0)$, $\boldsymbol{r_4} = (1.5858, 4.4142)$,
$\boldsymbol{r_5} = (3.0, 5.0)$, $\boldsymbol{r_6} = (4.4142, 4.4142)$, $\boldsymbol{r_7} = (5.0, 3.0)$, $\boldsymbol{r_8} = (4.4142, 1.5858)$.



(b) Input data

Array x and y, $n = 9$, $is = 1$, $ie = 9$ and $m = 20$.

(c) Main program

```
/*      C interface example for ASL_dgispr */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int nx;
    int is;
    int ie;
    int m;
    double *xo;
    double *yo;
    double *wk;
    int ierr;
    int i,nwk;
    FILE *fp;

    fp = fopen( "dgispr.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgispr ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=9;
    m=20;
    is=1;
    ie=9;

    nwk=9*nx+m-3;
    wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }

    y = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    xo = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( xo == NULL )
    {
        printf( "no enough memory for array xo\n" );
        return -1;
    }

    yo = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( yo == NULL )
    {
        printf( "no enough memory for array yo\n" );
        return -1;
    }

    printf( "\tn   = %6d\n", nx  );
    printf( "\tis  = %6d\n", is  );
    printf( "\tie  = %6d\n", ie  );
    printf( "\tm   = %6d\n", m   );
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &x[i] );
    }
    for( i=0 ; i<nx ; i++ )
    {
        fscanf( fp, "%lf", &y[i] );
    }

    printf( "\n" );
    printf( "\tCoordinates (x,y)\n\n" );
    printf( "\t     i               x[i]               y[i]\n");
    for( i=0 ; i<nx ; i++ )
    {
```

```
        printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],y[i] );
    }

    fclose( fp );

    ierr = ASL_dgispr(x, y, nx, is, ie, m, xo, yo, wk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\t     i               xo[i]              yo[i]\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf( "\t%6d           %8.3g           %8.3g\n", i,xo[i],yo[i] );
    }

    free( x );
    free( y );
    free( xo );
    free( yo );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgispr ***

 ** Input **

n   =      9
is  =      1
ie  =      9
m   =     20

Coordinates (x,y)

     i                x[i]               y[i]
     0                   3                  1
     1                1.59               1.59
     2                   1                  3
     3                1.59               4.41
     4                   3                  5
     5                4.41               4.41
     6                   5                  3
     7                4.41               1.59
     8                   3                  1

 ** Output **

ierr =      0

     i               xo[i]              yo[i]
     0                   3                  1
     1                2.35               1.11
     2                1.77               1.42
     3                1.33               1.91
     4                1.06               2.51
     5                1.01               3.16
     6                1.17                3.8
     7                1.53               4.35
     8                2.05               4.76
     9                2.67               4.97
    10                3.33               4.97
    11                3.95               4.76
    12                4.47               4.35
    13                4.83                3.8
    14                4.99               3.16
    15                4.94               2.51
    16                4.67               1.91
    17                4.23               1.42
    18                3.65               1.11
    19                   3                  1
```

6.4.3 ASL_dgisso, ASL_rgisso
Open Curve Smoothed Interpolation

(1) **Function**

ASL_dgisso or ASL_rgisso performs a plane data smoothed interpolation when the string of input data points takes the shape of an open curve. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgisso  (x, y, n, is, ie, m, xo, yo, wk);
```

Single precision:

```
ierr = ASL_rgisso  (x, y, n, is, ie, m, xo, yo, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]), i = 1, ··· , n (Input them in the order they are to be interpolated)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
3	n	I	1	Input	Number of sample points
4	is	I	1	Input	Sample point element number of interpolation starting point Interpolation starting point is (x[is − 1], y[is − 1])
5	ie	I	1	Input	Sample point element number of interpolation end point Interpolation end point is (x[ie − 1], y[ie − 1])
6	m	I	1	Input	Number of interpolation data values (=Number of subdivisions+1)
7	xo	{D∗ / R∗}	m	Output	Abscissa value xo[i − 1] of interpolated point
8	yo	{D∗ / R∗}	m	Output	Ordinate value yo[i − 1] of interpolated point
9	wk	{D∗ / R∗}	See Contents	Work	Work area **Size**: 2 × n × n + 9 × n + m − 3
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n $\geq$ 4

(b) m $\geq$ 2

(c) 1 $\leq$ is $\leq$ n

(d) 1 $\leq$ ie $\leq$ n

(e) x[0] $<$ x[1] $< \cdots <$ x[n $-$ 1] (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	X coordinate became to be out of interpolation interval.	The extrapolated value is output for points out of interpolation interval.
3000	Restriction (a), (b), (c) or (d) was not satisfied.	Processing is aborted.
3010	Restriction (e) was not satisfied.	
4000	The cross validation minimum value was not found (data is uncorrelated).	

(6) **Notes**

(a) Generally, obtained interpolation points are not equally spaced.

(7) **Example**

(a) Problem

$$\begin{cases} x_i = 7.5S_i^3 - 11.5S_i^2 + 5S_i + e_{xi} \\ y_i = -0.67S_i^3 - S_i^2 + 2S_i + e_{yi} \end{cases} \quad (i = 1, \cdots, 15)$$

$$S_i = 0.06 \times i$$

Assume that e_{xi} and e_{yi} are normal distribution random numbers having mean 0 and standard deviation 0.05.

(b) Input data
n=15, is=1, ie=10 and m=10.

(c) Main program

```
/*      C interface example for ASL_dgisso */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *x;
    double *y;
    int nx;
    int is;
    int ie;
    int m;
    double *xo;
    double *yo;
    double *wk;
    int ierr;
    double *rn;
    double  am;
    double  sg;
```

```
double  si;
double  esi;
int ix;
int iy;
int i,ni,nwk;

printf( "    *** ASL_dgisso ***\n" );
printf( "\n     ** Input **\n\n" );
nx=15;
m=10;
ix=1;
iy=1;
am=0.0;
sg=0.05;

nwk=(2*nx+9)*nx+m-3;
wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

x = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( x == NULL )
{
    printf( "no enough memory for array x\n" );
    return -1;
}

y = ( double * )malloc((size_t)( sizeof(double) * nx ));
if( y == NULL )
{
    printf( "no enough memory for array y\n" );
    return -1;
}

xo = ( double * )malloc((size_t)( sizeof(double) * m ));
if( xo == NULL )
{
    printf( "no enough memory for array xo\n" );
    return -1;
}

yo = ( double * )malloc((size_t)( sizeof(double) * m ));
if( yo == NULL )
{
    printf( "no enough memory for array yo\n" );
    return -1;
}

ni=50;
rn = ( double * )malloc((size_t)( sizeof(double) * ni ));
if( rn == NULL )
{
    printf( "no enough memory for array rn\n" );
    return -1;
}


ierr = ASL_djdbno( ni, am, sg, &ix, &iy, rn);
si=0.06;
esi=0.06;
for( i=0 ; i<15 ; i++ )
{
    x[i]=((7.5*si-11.5)*si+5.0)*si+rn[2*i];
    y[i]=((-0.67*si-1.0)*si+2.0)*si+rn[2*i+1];
    si += esi;
}
is=1;
ie=10;

printf( "\tn   = %6d\n", nx  );
printf( "\tis  = %6d\n", is  );
printf( "\tie  = %6d\n", ie  );
printf( "\tm   = %6d\n", m   );

printf( "\n" );
printf( "\tCoordinates (x,y)\n\n" );
printf( "\t     i               x[i]               y[i]\n");
for( i=0 ; i<nx ; i++ )
{
    printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],y[i] );
}

ierr = ASL_dgisso(x, y, nx, is, ie, m, xo, yo, wk);


printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n", ierr );
printf( "\n" );
```

```
    printf( "\t     i               xo[i]              yo[i]\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf( "\t%6d           %8.3g           %8.3g\n", i,xo[i],yo[i] );
    }

    free( x );
    free( y );
    free( xo );
    free( yo );
    free( wk );
    free( rn );

    return 0;
}
```

(d) Output results

```
*** ASL_dgisso ***

 ** Input **

n   =     15
is  =      1
ie  =     10
m   =     10

Coordinates (x,y)

     i                x[i]              y[i]
     0               0.164             0.131
     1               0.517             0.218
     2               0.517             0.336
     3               0.661             0.432
     4               0.574             0.455
     5               0.654             0.501
     6               0.648             0.692
     7               0.625             0.702
     8               0.478             0.761
     9               0.449              0.73
    10               0.505             0.628
    11                0.35             0.539
    12                0.44             0.628
    13               0.564             0.697
    14               0.646             0.439

 ** Output **

ierr =      0

     i               xo[i]              yo[i]
     0               0.181              0.13
     1               0.308             0.129
     2               0.424             0.165
     3               0.516             0.272
     4               0.583             0.389
     5               0.624             0.443
     6               0.636             0.502
     7               0.618             0.638
     8               0.564             0.744
     9               0.495             0.732
```

6.4.4 ASL_dgissr, ASL_rgissr
Closed Curve Smoothed Interpolation

(1) **Function**

ASL_dgissr or ASL_rgissr performs a plane data smoothed interpolation when the string of input data points takes the shape of a closed curve. Abscissa values of knots are set equal to abscissa values of sample points.

(2) **Usage**

Double precision:

ierr = ASL_dgissr (x, y, n, is, ie, m, xo, yo, wk);

Single precision:

ierr = ASL_rgissr (x, y, n, is, ie, m, xo, yo, wk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	x	{D∗ / R∗}	n	Input	Abscissa values x[i − 1] of sample point (x[i − 1], y[i − 1]), i = 1, ··· , n (Input them in the order they are to be interpolated)
2	y	{D∗ / R∗}	n	Input	Ordinate values of sample points or function values y[i − 1]
3	n	I	1	Input	Number of sample points
4	is	I	1	Input	Sample point element number of interpolation starting point Interpolation starting point is (x[is − 1], y[is − 1])
5	ie	I	1	Input	Sample point element number of interpolation end point Interpolation end point is (x[ie − 1], y[ie − 1])
6	m	I	1	Input	Number of interpolation data values (=Number of subdivisions+1)
7	xo	{D∗ / R∗}	m	Output	Abscissa value xo[i − 1] of interpolated point
8	yo	{D∗ / R∗}	m	Output	Ordinate value yo[i − 1] of interpolated point
9	wk	{D∗ / R∗}	See Contents	Work	Work area **Size:** 2 × n × n + 9 × n + m − 3
10	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $n \geq 4$

(b) $m \geq 2$

(c) $1 \leq is \leq n$

(d) $1 \leq ie \leq n$

(e) $x[0] < x[1] < \cdots < x[n-1]$ (Ascending order)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
1000	X coordinate became to be out of interpolation interval.	The extrapolated value is output for points out of interpolation interval.
3000	Restriction (a), (b), (c) or (d) was not satisfied.	Processing is aborted.
3010	Restriction (e) was not satisfied.	
4000	The cross validation minimum value was not found (data is uncorrelated).	

(6) **Notes**

(a) Generally, obtained interpolation points are not equally spaced.

(b) The point (x[0], y[0]) must be almost identical with $(x[n-1], y[n-1])$.

(7) **Example**

(a) Problem

$$\begin{cases} x_i = \sin(S_i) + e_{xi} \\ y_i = \cos(S_i) + e_{yi} \end{cases} \quad (i = 1, \cdots, 16)$$

$$S_i = 0.418879 \times (i-1)$$

Assume that e_{xi} and e_{yi} are normal distribution random numbers having mean 0 and standard deviation 0.05.

(b) Input data
n=16, is=1, ie=16 and m=12.

(c) Main program

```
/*      C interface example for ASL_dgissr */

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

int main()
{
    double *x;
    double *y;
    int nx;
    int is;
    int ie;
    int m;
    double *xo;
    double *yo;
    double *wk;
    int ierr;
    double *rn;
    double  am;
    double  sg;
    double  si;
    double  esi;
    int ix;
    int iy;
    int i,ni,nwk;

    printf( "    *** ASL_dgissr ***\n" );
    printf( "\n     ** Input **\n\n" );
    nx=16;
    m=12;
    ix=1;
    iy=1;
    am=0.0;
    sg=0.05;

    nwk=(2*nx+9)*nx+m-3;
    wk = ( double * )malloc((size_t)( sizeof(double) * nwk ));
    if( wk == NULL )
    {
        printf( "no enough memory for array wk\n" );
        return -1;
    }

    x = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( x == NULL )
    {
        printf( "no enough memory for array x\n" );
        return -1;
    }

    y = ( double * )malloc((size_t)( sizeof(double) * nx ));
    if( y == NULL )
    {
        printf( "no enough memory for array y\n" );
        return -1;
    }

    xo = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( xo == NULL )
    {
        printf( "no enough memory for array xo\n" );
        return -1;
    }

    yo = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( yo == NULL )
    {
        printf( "no enough memory for array yo\n" );
        return -1;
    }

    ni=40;
    rn = ( double * )malloc((size_t)( sizeof(double) * ni ));
    if( rn == NULL )
    {
        printf( "no enough memory for array rn\n" );
        return -1;
    }


    ierr = ASL_djdbno( ni, am, sg, &ix, &iy, rn);
    si=0.0;
    esi=0.418879;
    for( i=0 ; i<16 ; i++ )
    {
        x[i]=sin(si)+rn[2*i];
        y[i]=cos(si)+rn[2*i+1];
        si += esi;
    }
```

```
    is=1;
    ie=16;

    printf( "\tn   = %6d\n", nx  );
    printf( "\tis  = %6d\n", is  );
    printf( "\tie  = %6d\n", ie  );
    printf( "\tm   = %6d\n", m   );

    printf( "\n" );
    printf( "\tCoordinates (x,y)\n\n" );
    printf( "\t     i               x[i]               y[i]\n");
    for( i=0 ; i<nx ; i++ )
    {
        printf( "\t%6d           %8.3g           %8.3g\n", i,x[i],y[i] );
    }


    ierr = ASL_dgissr(x, y, nx, is, ie, m, xo, yo, wk);


    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );
    printf( "\n" );

    printf( "\t     i               xo[i]              yo[i]\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf( "\t%6d           %8.3g           %8.3g\n", i,xo[i],yo[i] );
    }

    free( x );
    free( y );
    free( xo );
    free( yo );
    free( wk );
    free( rn );

    return 0;
}
```

(d) Output results

```
*** ASL_dgissr ***

 ** Input **

n   =     16
is  =      1
ie  =     16
m   =     12

Coordinates (x,y)

     i               x[i]               y[i]
     0            -0.0959               1.01
     1              0.476              0.907
     2              0.689              0.681
     3               0.97              0.328
     4              0.901             -0.141
     5              0.861             -0.558
     6              0.609             -0.731
     7              0.253             -0.932
     8             -0.258               -0.9
     9             -0.618             -0.774
    10             -0.808             -0.563
    11              -1.08             -0.237
    12             -0.973              0.303
    13              -0.71              0.788
    14             -0.413              0.851
    15             0.0551              0.928

 ** Output **

ierr =      0

     i               xo[i]              yo[i]
     0             -0.019              0.981
     1               0.47               0.88
     2               0.86              0.476
     3              0.952            -0.0832
     4              0.754             -0.617
     5              0.279             -0.912
     6             -0.303             -0.896
     7             -0.804               -0.6
     8              -1.05              -0.11
     9             -0.927              0.463
    10             -0.523              0.831
    11             -0.019              0.981
```

6.5 B–SPLINE

6.5.1 ASL_dgicbs, ASL_rgicbs B-Spline Calculation

(1) **Function**

ASL_dgicbs or ASL_rgicbs calculates m nonzero B–spline values for x in the range $\xi_0 \leq x \leq \xi_{n+1}$ when the knots $\xi_{1-m}, \xi_{2-m}, \cdots, \xi_{n+m}$ and the B–spline order (degree+1) m are given.

(2) **Usage**

Double precision:

```
ierr = ASL_dgicbs  (xd, xk, n, m, aryn, &knot, wk);
```

Single precision:

```
ierr = ASL_rgicbs  (xd, xk, n, m, aryn, &knot, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{D / R}	1	Input	Sample points x.
2	xk	{D∗ / R∗}	n + 2 × m	Input	Knots (including additional knots) ξ_i.
3	n	I	1	Input	Number of internal knots n.
4	m	I	1	Input	B–spline order m.
5	aryn	{D∗ / R∗}	m	Output	Normalized B–spline $N_{mi}(x)$.
6	knot	I*	1	Output	Nonzero B–spline position.
7	wk	{D∗ / R∗}	m+1	Work	m–order B–spline $M_{mi}(x)$.
8	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) n ≥ 1

(b) m ≥ 1

(c) xk[m − 1] ≤ xd ≤ xk[m + n]

(d) xk[0] ≤ xk[1] ≤ ··· ≤ xk[n + 2 × m − 1]

(e) xk[i − 1] < xk[i + m − 1] (i = 1, 2, ··· , n + m)

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3300	Restriction (d) was not satisfied.	
3400	Restriction (e) was not satisfied.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(7) **Example**

(a) Problem
Given the knots $\xi_i = \{0, 0, 0, 0, 1, 2, 3, 5, 6, 7, 7, 7, 7\}$ $(i = -3, -2, \cdots, 9)$ and the B–spline order $m = 4$, obtain 4 nonzero B–spline values for $x = 4$.

(b) Input data
xd=4, knots xk, n=5 and m=4.

(c) Main Program

```
/*      C interface example for ASL_dgicbs */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double xd;
    double *xk;
    int n;
    int m;
    double *aryn;
    int knot;
    double *arym;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dgicbs.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgicbs ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &n );
    fscanf( fp, "%d", &m );
    fscanf( fp, "%lf", &xd );

    xk = ( double * )malloc((size_t)( sizeof(double) * (n+2*m) ));
    if( xk == NULL )
    {
        printf( "no enough memory for array xk\n" );
        return -1;
    }

    aryn = ( double * )malloc((size_t)( sizeof(double) * m ));
    if( aryn == NULL )
    {
        printf( "no enough memory for array aryn\n" );
        return -1;
    }

    arym = ( double * )malloc((size_t)( sizeof(double) * (m+1) ));
    if( arym == NULL )
    {
```

```
        printf( "no enough memory for array arym\n" );
        return -1;
    }

    printf( "\tn = %6d m = %6d\n", n, m );
    printf( "\txd = %8.3g\n", xd);

    for( i=0 ; i<n+2*m ; i++ )
    {
        fscanf( fp, "%lf", &xk[i] );
        printf( "\txk[%6d] = %8.3g\n", i, xk[i] );
    }

    fclose( fp );

    ierr = ASL_dgicbs(xd, xk, n, m, aryn, &knot, arym);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n\n", ierr );

    printf( "\tValues of normalized B-spline\n\n" );
    for( i=0 ; i<m ; i++ )
    {
        printf( "\taryn[%6d] = %8.3g\n", i, aryn[i] );
    }

    free( xk );
    free( aryn );
    free( arym );

    return 0;
}
```

(d) Output results

```
*** ASL_dgicbs ***

 ** Input **

n =      5 m =      4
xd =      0.4
xk[     0] =        0
xk[     1] =        0
xk[     2] =        0
xk[     3] =        0
xk[     4] =      0.1
xk[     5] =      0.2
xk[     6] =      0.3
xk[     7] =      0.5
xk[     8] =      0.6
xk[     9] =      0.7
xk[    10] =      0.7
xk[    11] =      0.7
xk[    12] =      0.7

 ** Output **

ierr =      0

Values of normalized B-spline

aryn[     0] =   0.0417
aryn[     1] =    0.458
aryn[     2] =    0.458
aryn[     3] =   0.0417
```

6.5.2 ASL_dgisi1, ASL_rgisi1
Interpolation Using a B-Spline (One-Dimensional Data)

(1) **Function**

ASL_dgisi1 or ASL_rgisi1 obtains the following spline function of degree $(m-1)$ for interpolating the data $f_i \ \ (i = 1, 2, \cdots, N)$ at the sample points $x_i \ \ (i = 1, 2, \cdots, N)$:

$$S(x) \quad = \quad \sum_{i=1}^{n+m} c_i N_{mi}(x)$$

(2) **Usage**

Double precision:

ierr = ASL_dgisi1 (xd, nd, fd, xk, m, xx, nn, s, wk, iwk);

Single precision:

ierr = ASL_rgisi1 (xd, nd, fd, xk, m, xx, nn, s, wk, iwk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{ D∗ / R∗ }	nd	Input	Sample points x_i.
2	nd	I	1	Input	Number of sample points N.
3	fd	{ D∗ / R∗ }	nd	Input	Data f_i given at sample points x_i.
4	xk	{ D∗ / R∗ }	nd + m	Input	Knots (including additional knots) ξ_i.
5	m	I	1	Input	B–spline order m.
6	xx	{ D∗ / R∗ }	nn	Input	x coordinates for calculating interpolation values.
7	nn	I	1	Input	Number of points for calculating interpolation values.
8	s	{ D∗ / R∗ }	nn	Output	Approximation function value $S(x)$ at $x =$ xx[i − 1] (i = 1, 2, ⋯, nn).
9	wk	{ D∗ / R∗ }	See Contents	Work	Work area. **Size**:$\mathrm{nd}^2 + \mathrm{nd} + 2 \times \mathrm{m} + 1$
10	iwk	I*	nd	Work	Work area.
11	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{nd} \geq 1$

(b) $\mathrm{nd} - \mathrm{m} \geq 1, \mathrm{m} \geq 1$

(c) $\mathrm{nn} \geq 1$

(d) $\mathrm{xd}[0] < \mathrm{xd}[1] < \cdots < \mathrm{xd}[\mathrm{nd}-1]$

(e) $\mathrm{xk}[\mathrm{m}-1] \leq \mathrm{xd}[\mathrm{i}-1] \leq \mathrm{xk}[\mathrm{nd}] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nd})$

(f) $\mathrm{xk}[0] \leq \mathrm{xk}[1] \leq \cdots \leq \mathrm{xk}[\mathrm{nd}+\mathrm{m}-1]$

(g) $\mathrm{xk}[\mathrm{i}-1] < \mathrm{xk}[\mathrm{i}+\mathrm{m}-1] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nd})$

(h) The sample values $\mathrm{xd}[\mathrm{i}-1] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(i) $\mathrm{xk}[\mathrm{m}-1] \leq \mathrm{xx}[\mathrm{i}-1] \leq \mathrm{xk}[\mathrm{nd}] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nn})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
2100	When the normalized equation is solved, there existed the diagonal element which was close to zero in the LU decomposition. The precise solutions may not be obtained.	Processing continues.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3400	Restriction (d) was not satisfied.	
3500	Restriction (e) was not satisfied.	
3600	Restriction (f) was not satisfied.	
3700	Restriction (g) was not satisfied.	
3800	Restriction (h) was not satisfied.	
3900	Restriction (i) was not satisfied.	
4000	The normalized equation could not be solved.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(b) Number of internal knots is $\mathrm{nd} - \mathrm{m}$.

(7) **Example**

(a) Problem

Obtain interpolation values for $x : 0.05, 0.10, \cdots, 0.95$ by interpolating the following data:

$$f_i = \frac{1}{0.01 + (x_i - 0.3)^2} \quad (x_i = 0.0, 0.1, \cdots, 1.0)$$

(b) Input data

Sample points (xd, fd), nd=11, knots xk, m=4, x coordinates for calculating interpolation values xx and nn=19.

(c) Main Program

```
/*      C interface example for ASL_dgisi1 */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *xd;
    int nd;
    double *fd;
    double *xk;
    int m;
    double *xx;
    int nn;
    double *s;
    double *wk;
    int *iwk;
    int ierr;
    int i;
    FILE *fp;

    fp = fopen( "dgisi1.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgisi1 ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &nd );
    fscanf( fp, "%d", &m );
    fscanf( fp, "%d", &nn );

    iwk = ( int * )malloc((size_t)( sizeof(int) * nd ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    xd = ( double * )malloc((size_t)( sizeof(double) * nd ));
    if( xd == NULL )
    {
        printf( "no enough memory for array xd\n" );
        return -1;
    }

    fd = ( double * )malloc((size_t)( sizeof(double) * nd ));
    if( fd == NULL )
    {
        printf( "no enough memory for array fd\n" );
        return -1;
    }

    xk = ( double * )malloc((size_t)( sizeof(double) * (nd+m) ));
    if( xk == NULL )
    {
        printf( "no enough memory for array xk\n" );
        return -1;
    }

    xx = ( double * )malloc((size_t)( sizeof(double) * nn ));
    if( xx == NULL )
    {
        printf( "no enough memory for array xx\n" );
        return -1;
    }

    s = ( double * )malloc((size_t)( sizeof(double) * nn ));
    if( s == NULL )
```

```
{
    printf( "no enough memory for array s\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double)
    * (nd*nd+nd+2*m+1) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tnd = %6d m = %6d nn = %6d\n",nd,m,nn );

printf( "\n\n\txd\n\n");
for( i=0 ; i<nd ; i++ )
{
    xd[i] = 0.1*(double)i;
    printf( "\txd[%6d] = %8.3g\n", i, xd[i] );
    fd[i] = 1.0/(0.01+(xd[i]-0.3)*(xd[i]-0.3));
}

printf( "\n\n\txk\n\n");
for( i=0 ; i<nd+m ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
    printf( "\txk[%6d] = %8.3g\n", i, xk[i] );
}

for( i=0 ; i<nn ; i++ )
{
    xx[i] = 0.05*(double)(i+1);
}


fclose( fp );

ierr = ASL_dgisi1(xd, nd, fd, xk, m, xx, nn, s, wk, iwk);

printf( "\n     ** Output **\n\n" );
printf( "\tierr = %6d\n\n", ierr );

printf( "\tValues of approximating spline function on sample points\n\n" );
printf( "\t       xx  \t       s\n\n" );
for( i=0 ; i<nn ; i++ )
{
    printf( "\t%8.3g\t%8.3g\n", xx[i], s[i] );
}

free( iwk );
free( xd );
free( fd );
free( xk );
free( xx );
free( s );
free( wk );

return 0;
}
```

(d) Output results

```
*** ASL_dgisi1 ***

 ** Input **

nd =     11 m =      4 nn =     19


xd

xd[     0] =        0
xd[     1] =      0.1
xd[     2] =      0.2
xd[     3] =      0.3
xd[     4] =      0.4
xd[     5] =      0.5
xd[     6] =      0.6
xd[     7] =      0.7
xd[     8] =      0.8
xd[     9] =      0.9
xd[    10] =        1


xk

xk[     0] =        0
xk[     1] =        0
xk[     2] =        0
xk[     3] =        0
xk[     4] =      0.2
xk[     5] =      0.3
```

```
xk[     6] =      0.4
xk[     7] =      0.5
xk[     8] =      0.6
xk[     9] =      0.7
xk[    10] =      0.8
xk[    11] =        1
xk[    12] =        1
xk[    13] =        1
xk[    14] =        1

 ** Output **

ierr =      0

Values of approximating spline function on sample points

       xx          s

    0.05        15.7
     0.1          20
    0.15        29.3
     0.2          50
    0.25        82.2
     0.3         100
    0.35          82
     0.4          50
    0.45        29.7
     0.5          20
    0.55          14
     0.6          10
    0.65        7.48
     0.7        5.88
    0.75        4.72
     0.8        3.85
    0.85        3.19
     0.9         2.7
    0.95        2.32
```

6.5.3 ASL_dgisi2, ASL_rgisi2
Interpolation Using a B-Spline (Two-Dimensional Data)

(1) **Function**

ASL_dgisi2 or ASL_rgisi2 obtains the following spline function for interpolating the data f_{ij} $(i = 1, 2, \cdots, N_x; j = 1, 2, \cdots, N_y)$ at the sample points (x_i, y_j) $(i = 1, 2, \cdots, N_x; j = 1, 2, \cdots, N_y)$:

$$S(x, y) = \sum_{i=1}^{h+m} \sum_{j=1}^{k+n} c_{ij} N_{mi}(x) N_{nj}(y)$$

and calculates the interpolation values at the specified grid points.

(2) **Usage**

Double precision:

ierr = ASL_dgisi2 (xd, nxd, yd, nyd, fd, xk, mx, yk, my, xx, nnx, yy, nny, s, wk, iwk);

Single precision:

ierr = ASL_rgisi2 (xd, nxd, yd, nyd, fd, xk, mx, yk, my, xx, nnx, yy, nny, s, wk, iwk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{D∗ / R∗}	nxd	Input	Sample points in x direction x_i.
2	nxd	I	1	Input	Number of sample points in x direction N_x.
3	yd	{D∗ / R∗}	nyd	Input	Sample points in y direction y_j.
4	nyd	I	1	Input	Number of sample points in y direction N_y.
5	fd	{D∗ / R∗}	nxd×nyd	Input	Data f_{ij} given at sample points (x_i, y_j).
6	xk	{D∗ / R∗}	nxd + mx	Input	Knots (including additional knots) in x direction ξ_i.
7	mx	I	1	Input	x–direction B–spline order m.
8	yk	{D∗ / R∗}	nyd + my	Input	Knots (including additional knots) in y direction ζ_j.
9	my	I	1	Input	y–direction B–spline order n.
10	xx	{D∗ / R∗}	nnx	Input	x coordinates for calculating interpolation values.
11	nnx	I	1	Input	Number of x–direction points for calculating interpolation values.
12	yy	{D∗ / R∗}	nny	Input	y coordinates for calculating interpolation values.
13	nny	I	1	Input	Number of y–direction points for calculating interpolation values.
14	s	{D∗ / R∗}	nnx×nny	Output	Approximation function value $S(x, y)$ at $(x, y) = (\mathrm{xx[i-1]}, \mathrm{yy[j-1]})$ $(\mathrm{i} = 1, 2, \cdots, \mathrm{nnx}\ ;\ \mathrm{j} = 1, 2, \cdots, \mathrm{nny})$.
15	wk	{D∗ / R∗}	See Contents	Work	Work area. **Size**: $\mathrm{nxd}^2 \times \mathrm{nyd}^2 + \mathrm{nxd} \times (\mathrm{nyd} + \mathrm{mx}) + 2 \times \mathrm{mx} + 2 \times \mathrm{my} + 2$
16	iwk	I*	See Contents	Work	Work area. **Size**: $\mathrm{nxd} \times \mathrm{nyd} + \mathrm{nxd} + \mathrm{nyd}$
17	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{nxd} \geq 1, \mathrm{nyd} \geq 1$

(b) $\mathrm{nxd} - \mathrm{mx} \geq 1, \mathrm{nyd} - \mathrm{my} \geq 1, \mathrm{mx} \geq 1, \mathrm{my} \geq 1$

(c) $\mathrm{nnx} \geq 1, \mathrm{nny} \geq 1$

(d) $\mathrm{xd}[0] < \mathrm{xd}[1] < \cdots < \mathrm{xd}[\mathrm{nxd} - 1]$

(e) $\mathrm{yd}[0] < \mathrm{yd}[1] < \cdots < \mathrm{yd}[\mathrm{nyd} - 1]$

(f) $\mathrm{xk}[\mathrm{mx} - 1] \leq \mathrm{xd}[\mathrm{i} - 1] \leq \mathrm{xk}[\mathrm{nxd}] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nxd})$

(g) $\mathrm{yk}[\mathrm{my} - 1] \leq \mathrm{yd}[\mathrm{j} - 1] \leq \mathrm{yk}[\mathrm{nyd}] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nyd})$

(h) $\mathrm{xk}[0] \leq \mathrm{xk}[1] \leq \cdots \leq \mathrm{xk}[\mathrm{nxd} + \mathrm{mx} - 1]$

(i) $\mathrm{yk}[0] \leq \mathrm{yk}[1] \leq \cdots \leq \mathrm{yk}[\mathrm{nyd} + \mathrm{my} - 1]$

(j) $\mathrm{xk}[\mathrm{i} - 1] < \mathrm{xk}[\mathrm{i} + \mathrm{mx} - 1] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nxd})$

(k) $\mathrm{yk}[\mathrm{j} - 1] < \mathrm{yk}[\mathrm{j} + \mathrm{my} - 1] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nyd})$

(l) The sample values $\mathrm{xd}[\mathrm{i} - 1] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nxd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(m) The sample values $\mathrm{yd}[\mathrm{j} - 1] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nyd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(n) $\mathrm{xk}[\mathrm{mx} - 1] \leq \mathrm{xx}[\mathrm{i} - 1] \leq \mathrm{xk}[\mathrm{nxd}] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nnx})$

(o) $\mathrm{yk}[\mathrm{my} - 1] \leq \mathrm{yy}[\mathrm{j} - 1] \leq \mathrm{yk}[\mathrm{nyd}] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nny})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
2100	When the normalized equation is solved, there existed the diagonal element which was close to zero in the LU decomposition. The precise solutions may not be obtained.	Processing continues.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3400	Restriction (d) was not satisfied.	
3410	Restriction (e) was not satisfied.	
3500	Restriction (f) was not satisfied.	
3510	Restriction (g) was not satisfied.	
3600	Restriction (h) was not satisfied.	
3610	Restriction (i) was not satisfied.	
3700	Restriction (j) was not satisfied.	
3710	Restriction (k) was not satisfied.	
3800	Restriction (l) was not satisfied.	
3810	Restriction (m) was not satisfied.	
3900	Restriction (n) was not satisfied.	
3910	Restriction (o) was not satisfied.	
4000	The normalized equation could not be solved.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(b) Number of internal knots is $(\mathrm{nxd} - \mathrm{mx}, \mathrm{nyd} - \mathrm{my})$.

(7) **Example**

(a) Problem

Obtain interpolation values for $x : 0.0, 0.1, \cdots, 0.9; y : 0.0, 0.1, \cdots, 0.9$ by interpolating the following data:

$$f_{ij} = \frac{1}{0.01 + (x_i - 0.3)^2} + \frac{1}{0.02 + (y_j - 0.4)^2}$$
$$(x_i = 0.0, 0.1, \cdots, 1.0; y_j = 0.0, 0.1, \cdots, 1.0)$$

(b) Input data

Sample points (xd, yd, fd), nxd=11, nyd=11, knots in x direction xk, mx=4, knots in y direction yk, my=4, x coordinates for calculating interpolation values xx, nnx=10, y coordinates for calculating interpolation values yy and nny=10.

(c) Main Program

```
/*      C interface example for ASL_dgisi2 */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *xd;
    int nxd;
    double *yd;
    int nyd;
    double *fd;
    double *xk;
    int mx;
    double *yk;
    int my;
    double *xx;
    int nnx;
    double *yy;
    int nny;
    double *s;
    double *wk;
    int *iwk;
    int ierr;
    int i,j;
    FILE *fp;

    fp = fopen( "dgisi2.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgisi2 ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &nxd );
    fscanf( fp, "%d", &mx );
    fscanf( fp, "%d", &nnx );
    fscanf( fp, "%d", &nyd );
    fscanf( fp, "%d", &my );
    fscanf( fp, "%d", &nny );

    iwk = ( int * )malloc((size_t)( sizeof(int)
        * (nxd*nyd+nxd+nyd) ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    xd = ( double * )malloc((size_t)( sizeof(double) * nxd ));
    if( xd == NULL )
    {
        printf( "no enough memory for array xd\n" );
        return -1;
    }

    yd = ( double * )malloc((size_t)( sizeof(double) * nyd ));
    if( yd == NULL )
    {
        printf( "no enough memory for array yd\n" );
        return -1;
```

```
}

fd = ( double * )malloc((size_t)( sizeof(double)*nxd*nyd ));
if( fd == NULL )
{
    printf( "no enough memory for array fd\n" );
    return -1;
}

xk = ( double * )malloc((size_t)( sizeof(double) * (nxd+mx) ));
if( xk == NULL )
{
    printf( "no enough memory for array xk\n" );
    return -1;
}

yk = ( double * )malloc((size_t)( sizeof(double) * (nyd+my) ));
if( yk == NULL )
{
    printf( "no enough memory for array yk\n" );
    return -1;
}

xx = ( double * )malloc((size_t)( sizeof(double) * nnx ));
if( xx == NULL )
{
    printf( "no enough memory for array xx\n" );
    return -1;
}

yy = ( double * )malloc((size_t)( sizeof(double) * nny ));
if( yy == NULL )
{
    printf( "no enough memory for array yy\n" );
    return -1;
}

s = ( double * )malloc((size_t)( sizeof(double) * nnx*nny ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double)
    * (nxd*nxd*nyd*nyd+nxd*(nyd+mx)
       +mx*2+my*2+2) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tnxd = %6d mx = %6d nnx = %6d\n",
    nxd, mx, nnx);
printf( "\tnyd = %6d my = %6d nny = %6d\n",
    nyd, my, nny);

printf( "\n\txd\n\n" );
for( i=0 ; i<nxd ; i++ )
{
    xd[i] = 0.1 * (double)i;
    printf( "\txd[%6d] = %8.3g\n", i, xd[i] );
}

printf( "\n\tyd\n\n" );
for( j=0 ; j<nyd ; j++ )
{
    yd[j] = 0.1 * (double)j;
    printf( "\tyd[%6d] = %8.3g\n", j, yd[j] );
}

for( j=0 ; j<nyd ; j++ )
{
    for( i=0 ; i<nxd ; i++ )
    {
        fd[i+nxd*j]=
         1.0/(0.01+(xd[i]-0.3)*(xd[i]-0.3))
        +1.0/(0.02+(yd[j]-0.4)*(yd[j]-0.4));
    }
}

printf( "\n\txk\n\n" );
for( i=0 ; i<nxd+mx ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
    printf( "\txk[%6d] = %8.3g\n", i, xk[i] );
}

printf( "\n\tyk\n\n" );
```

```
    for( j=0 ; j<nyd+my ; j++ )
    {
        fscanf( fp, "%lf", &yk[j] );
        printf( "\tyk[%6d] = %8.3g\n", j, yk[j] );
    }

    for( i=0 ; i<nnx ; i++ )
    {
        xx[i] = 0.1 * (double)i;
    }

    for( j=0 ; j<nny ; j++ )
    {
        yy[j] = 0.1 * (double)j;
    }

    fclose( fp );

    ierr = ASL_dgisi2(xd, nxd, yd, nyd, fd, xk, mx, yk, my, xx, nnx, yy,
             nny, s, wk, iwk);

    printf( "\n     ** Output **\n\n" );
    printf( "\tierr = %6d\n", ierr );

    printf( "\n\n\tValues of an approximating spline function on sample points\n\n" );
    printf( "\n\n\t      xx   \t      yy   \t     s\n\n" );
    for( i=0 ; i<nnx ; i++ )
    {
        for( j=0 ; j<nny ; j++ )
        {
            printf( "\t%8.3g\t%8.3g\t%8.3g\n",
            xx[i], yy[j], s[i+nnx*j] );
        }
    }

    free( iwk );
    free( xd );
    free( yd );
    free( fd );
    free( xk );
    free( yk );
    free( xx );
    free( yy );
    free( s );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgisi2 ***

 ** Input **

nxd =     11 mx =      4 nnx =     10
nyd =     11 my =      4 nny =     10

xd

xd[     0] =        0
xd[     1] =      0.1
xd[     2] =      0.2
xd[     3] =      0.3
xd[     4] =      0.4
xd[     5] =      0.5
xd[     6] =      0.6
xd[     7] =      0.7
xd[     8] =      0.8
xd[     9] =      0.9
xd[    10] =        1

yd

yd[     0] =        0
yd[     1] =      0.1
yd[     2] =      0.2
yd[     3] =      0.3
yd[     4] =      0.4
yd[     5] =      0.5
yd[     6] =      0.6
yd[     7] =      0.7
yd[     8] =      0.8
yd[     9] =      0.9
yd[    10] =        1

xk

xk[     0] =        0
```

```
xk[     1] =        0
xk[     2] =        0
xk[     3] =        0
xk[     4] =      0.2
xk[     5] =      0.3
xk[     6] =      0.4
xk[     7] =      0.5
xk[     8] =      0.6
xk[     9] =      0.7
xk[    10] =      0.8
xk[    11] =        1
xk[    12] =        1
xk[    13] =        1
xk[    14] =        1

yk

yk[     0] =        0
yk[     1] =        0
yk[     2] =        0
yk[     3] =        0
yk[     4] =      0.2
yk[     5] =      0.3
yk[     6] =      0.4
yk[     7] =      0.5
yk[     8] =      0.6
yk[     9] =      0.7
yk[    10] =      0.8
yk[    11] =        1
yk[    12] =        1
yk[    13] =        1
yk[    14] =        1

 ** Output **

ierr =      0

Values of an approximating spline function on sample points


      xx          yy          s

       0           0        15.6
       0         0.1        19.1
       0         0.2        26.7
       0         0.3        43.3
       0         0.4          60
       0         0.5        43.3
       0         0.6        26.7
       0         0.7        19.1
       0         0.8        15.6
       0         0.9        13.7
     0.1           0        25.6
     0.1         0.1        29.1
     0.1         0.2        36.7
     0.1         0.3        53.3
     0.1         0.4          70
     0.1         0.5        53.3
     0.1         0.6        36.7
     0.1         0.7        29.1
     0.1         0.8        25.6
     0.1         0.9        23.7
     0.2           0        55.6
     0.2         0.1        59.1
     0.2         0.2        66.7
     0.2         0.3        83.3
     0.2         0.4         100
     0.2         0.5        83.3
     0.2         0.6        66.7
     0.2         0.7        59.1
     0.2         0.8        55.6
     0.2         0.9        53.7
     0.3           0         106
     0.3         0.1         109
     0.3         0.2         117
     0.3         0.3         133
     0.3         0.4         150
     0.3         0.5         133
     0.3         0.6         117
     0.3         0.7         109
     0.3         0.8         106
     0.3         0.9         104
     0.4           0        55.6
     0.4         0.1        59.1
     0.4         0.2        66.7
     0.4         0.3        83.3
     0.4         0.4         100
     0.4         0.5        83.3
     0.4         0.6        66.7
     0.4         0.7        59.1
```

```
0.4         0.8         55.6
0.4         0.9         53.7
0.5           0         25.6
0.5         0.1         29.1
0.5         0.2         36.7
0.5         0.3         53.3
0.5         0.4           70
0.5         0.5         53.3
0.5         0.6         36.7
0.5         0.7         29.1
0.5         0.8         25.6
0.5         0.9         23.7
0.6           0         15.6
0.6         0.1         19.1
0.6         0.2         26.7
0.6         0.3         43.3
0.6         0.4           60
0.6         0.5         43.3
0.6         0.6         26.7
0.6         0.7         19.1
0.6         0.8         15.6
0.6         0.9         13.7
0.7           0         11.4
0.7         0.1           15
0.7         0.2         22.5
0.7         0.3         39.2
0.7         0.4         55.9
0.7         0.5         39.2
0.7         0.6         22.5
0.7         0.7           15
0.7         0.8         11.4
0.7         0.9         9.59
0.8           0          9.4
0.8         0.1         12.9
0.8         0.2         20.5
0.8         0.3         37.2
0.8         0.4         53.8
0.8         0.5         37.2
0.8         0.6         20.5
0.8         0.7         12.9
0.8         0.8          9.4
0.8         0.9         7.55
0.9           0         8.26
0.9         0.1         11.8
0.9         0.2         19.4
0.9         0.3           36
0.9         0.4         52.7
0.9         0.5           36
0.9         0.6         19.4
0.9         0.7         11.8
0.9         0.8         8.26
0.9         0.9         6.41
```

6.5.4 ASL_dgisi3, ASL_rgisi3
Interpolation Using a B-Spline (Three-Dimensional Data)

(1) **Function**

ASL_dgisi3 or ASL_rgisi3 obtains the following spline function for interpolating the data f_{ijk} $(i = 1, 2, \cdots, N_x;\ j = 1, 2, \cdots, N_y;\ k = 1, 2, \cdots, N_z)$ at the sample points (x_i, y_j, z_k) $(i = 1, 2, \cdots, N_x;\ j = 1, 2, \cdots, N_y;\ k = 1, 2, \cdots, N_z)$:

$$S(x,y,z) \quad = \quad \sum_{i=1}^{n_x+m_x} \sum_{j=1}^{n_y+m_y} \sum_{k=1}^{n_z+m_z} c_{ijk} N_{m_x i}(x) N_{m_y j}(y) N_{m_z k}(z)$$

and calculates the interpolation values at the specified grid points.

(2) **Usage**

Double precision:

ierr = ASL_dgisi3 (xd, nxd, yd, nyd, zd, nzd, fd, xk, mx, yk, my, zk, mz, xx, nnx, yy, nny, zz, nnz, s, wk, iwk);

Single precision:

ierr = ASL_rgisi3 (xd, nxd, yd, nyd, zd, nzd, fd, xk, mx, yk, my, zk, mz, xx, nnx, yy, nny, zz, nnz, s, wk, iwk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{D∗ / R∗}	nxd	Input	Sample points in x direction x_i.
2	nxd	I	1	Input	Number of sample points in x direction N_x.
3	yd	{D∗ / R∗}	nyd	Input	Sample points in y direction y_j.
4	nyd	I	1	Input	Number of sample points in y direction N_y.
5	zd	{D∗ / R∗}	nzd	Input	Sample points in z direction z_k.
6	nzd	I	1	Input	Number of sample points in z direction N_z.
7	fd	{D∗ / R∗}	See Contents	Input	Data f_{ijk} given at sample points (x_i, y_j, z_k). **Size**: nxd×nyd×nzd

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	xk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nxd + mx	Input	Knots (including additional knots) in x direction ξ_i.
9	mx	I	1	Input	x–direction B–spline order m_x.
10	yk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nyd + my	Input	Knots (including additional knots) in y direction ζ_j.
11	my	I	1	Input	y–direction B–spline order m_y.
12	zk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nzd + mz	Input	Knots (including additional knots) in z direction η_k.
13	mz	I	1	Input	z–direction B–spline order m_z.
14	xx	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nnx	Input	x coordinates for calculating interpolation values.
15	nnx	I	1	Input	Number of x–direction points for calculating interpolation values.
16	yy	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nny	Input	y coordinates for calculating interpolation values.
17	nny	I	1	Input	Number of y–direction points for calculating interpolation values.
18	zz	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nnz	Input	z coordinates for calculating interpolation values.
19	nnz	I	1	Input	Number of z–direction points for calculating interpolation value.
20	s	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Output	Approximation function value $S(x, y, z)$ at $(x, y, z) = (\text{xx[i}-1], \text{yy[j}-1], \text{zz[i}-1])$ $(\text{i} = 1, 2, \cdots, \text{nnx}\ ;\ \text{j} = 1, 2, \cdots, \text{nny};$ $\text{k} = 1, 2, \cdots, \text{nnz})$. **Size**: nnx×nny×nnz
21	wk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area. **Size**: $\text{nxd}^2 \times \text{nyd}^2 \times \text{nzd}^2 + \text{nxd} \times \text{nyd} \times \text{nzd} + \text{mx} \times (\text{nxd} + 2) + \text{my} \times (\text{nyd} + 2) + 2 \times \text{mz} + 3$
22	iwk	I*	See Contents	Work	Work area. **Size**: nxd × nyd × nzd + nxd + nyd + nzd
23	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{nxd} \geq 1, \text{nyd} \geq 1, \text{nzd} \geq 1$

(b) $\text{nxd} - \text{mx} \geq 1, \text{nyd} - \text{my} \geq 1, \text{nzd} - \text{mz} \geq 1, \text{mx} \geq 1, \text{my} \geq 1, \text{mz} \geq 1$

(c) $\text{nnx} \geq 1, \text{nny} \geq 1, \text{nnz} \geq 1$

(d) $\text{xd}[0] < \text{xd}[1] < \cdots < \text{xd}[\text{nxd} - 1]$

(e) $\text{yd}[0] < \text{yd}[1] < \cdots < \text{yd}[\text{nyd} - 1]$

(f) $zd[0] < zd[1] < \cdots < zd[nzd-1]$

(g) $xk[mx-1] \leq xd[i-1] \leq xk[nxd]$ $(i = 1, 2, \cdots, nxd)$

(h) $yk[my-1] \leq yd[j-1] \leq yk[nyd]$ $(j = 1, 2, \cdots, nyd)$

(i) $zk[mz-1] \leq zd[k-1] \leq zk[nzd]$ $(k = 1, 2, \cdots, nzd)$

(j) $xk[0] \leq xk[1] \leq \cdots \leq xk[nxd+mx-1]$

(k) $yk[0] \leq yk[1] \leq \cdots \leq yk[nyd+my-1]$

(l) $zk[0] \leq zk[1] \leq \cdots \leq zk[nzd+mz-1]$

(m) $xk[i-1] < xk[i+mx-1]$ $(i = 1, 2, \cdots, nxd)$

(n) $yk[j-1] < yk[j+my-1]$ $(j = 1, 2, \cdots, nyd)$

(o) $zk[k-1] < zk[k+mz-1]$ $(k = 1, 2, \cdots, nzd)$

(p) The sample values $xd[i-1]$ $(i = 1, 2, \cdots, nxd)$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(q) The sample values $yd[j-1]$ $(j = 1, 2, \cdots, nyd)$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(r) The sample values $zd[k-1]$ $(k = 1, 2, \cdots, nzd)$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(s) $xk[mx-1] \leq xx[i-1] \leq xk[nxd]$ $(i = 1, 2, \cdots, nnx)$

(t) $yk[my-1] \leq yy[j-1] \leq yk[nyd]$ $(j = 1, 2, \cdots, nny)$

(u) $zk[mz-1] \leq zz[k-1] \leq zk[nzd]$ $(k = 1, 2, \cdots, nnz)$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3400	Restriction (d) was not satisfied.	
3410	Restriction (e) was not satisfied.	
3420	Restriction (f) was not satisfied.	
3500	Restriction (g) was not satisfied.	
3510	Restriction (h) was not satisfied.	
3520	Restriction (i) was not satisfied.	
3600	Restriction (j) was not satisfied.	
3610	Restriction (k) was not satisfied.	
3620	Restriction (l) was not satisfied.	
3700	Restriction (m) was not satisfied.	
3710	Restriction (n) was not satisfied.	
3720	Restriction (o) was not satisfied.	
3800	Restriction (p) was not satisfied.	
3810	Restriction (q) was not satisfied.	
3820	Restriction (r) was not satisfied.	
3900	Restriction (s) was not satisfied.	

ierr value	Meaning	Processing
3910	Restriction (t) was not satisfied.	Processing is aborted.
3920	Restriction (u) was not satisfied.	
4000	The normalized equation could not be solved.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(b) Number of internal knots is $(\mathrm{nxd}-\mathrm{mx}, \mathrm{nyd}-\mathrm{my}, \mathrm{nzd}-\mathrm{mz})$.

(7) **Example**

(a) Problem

Obtain interpolation values for $x : 0.0, 0.1, \cdots, 0.8; y : 0.0, 0.1, \cdots, 0.8; z : 0.0, 0.1, \cdots, 0.8$ by interpolating the following data:

$$f_{ijk} = 10 + \frac{1}{0.03 + (x_i - 0.5)^2} + \frac{1}{0.04 + (y_j - 0.2)^2} + \frac{1}{0.05 + (z_k - 0.6)^2}$$
$$(x_i = 0.0, 0.1, \cdots, 0.8; y_j = 0.0, 0.1, \cdots, 0.8; z_k = 0.0, 0.1, \cdots, 0.8)$$

(b) Input data

Sample points (xd, yd, zd, fd), nxd=9, nyd=9, nzd=9, knots in x direction xk, mx=4, knots in y direction yk, my=4, knots in z direction zk, mz=4, x coordinates for calculating interpolation values xx, nnx=9, y coordinates for calculating interpolation values yy, nny=9, z coordinates for calculating interpolation values zz and nnz=9.

(c) Main Program

```
/*      C interface example for ASL_dgisi3 */

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

int main()
{
    double *xd;
    int nxd;
    double *yd;
    int nyd;
    double *zd;
    int nzd;
    double *fd;
    double *xk;
    int mx;
    double *yk;
    int my;
    double *zk;
    int mz;
    double *xx;
    int nnx;
    double *yy;
    int nny;
    double *zz;
    int nnz;
    double *s;
    double *wk;
    int *iwk;
    int ierr;
    double ff,sumt2,sumd2,fit;
    int i,j,k;
    FILE *fp;

    fp = fopen( "dgisi3.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgisi3 ***\n" );
```

```
printf( "\n     ** Input **\n\n" );

fscanf( fp, "%d", &nxd );
fscanf( fp, "%d", &mx );
fscanf( fp, "%d", &nnx );
fscanf( fp, "%d", &nyd );
fscanf( fp, "%d", &my );
fscanf( fp, "%d", &nny );
fscanf( fp, "%d", &nzd );
fscanf( fp, "%d", &mz );
fscanf( fp, "%d", &nnz );

iwk = ( int * )malloc((size_t)( sizeof(int) *
    (nxd*nyd*nzd+nxd+nyd+nzd) ));
if( iwk == NULL )
{
    printf( "no enough memory for array iwk\n" );
    return -1;
}

xd = ( double * )malloc((size_t)( sizeof(double) * nxd ));
if( xd == NULL )
{
    printf( "no enough memory for array xd\n" );
    return -1;
}

yd = ( double * )malloc((size_t)( sizeof(double) * nyd ));
if( yd == NULL )
{
    printf( "no enough memory for array yd\n" );
    return -1;
}

zd = ( double * )malloc((size_t)( sizeof(double) * nzd ));
if( zd == NULL )
{
    printf( "no enough memory for array zd\n" );
    return -1;
}

fd = ( double * )malloc((size_t)( sizeof(double) *
      nxd*nyd*nzd ));
if( fd == NULL )
{
    printf( "no enough memory for array fd\n" );
    return -1;
}

xk = ( double * )malloc((size_t)( sizeof(double) * (nxd+mx) ));
if( xk == NULL )
{
    printf( "no enough memory for array xk\n" );
    return -1;
}

yk = ( double * )malloc((size_t)( sizeof(double) * (nyd+my) ));
if( yk == NULL )
{
    printf( "no enough memory for array yk\n" );
    return -1;
}

zk = ( double * )malloc((size_t)( sizeof(double) * (nzd+mz) ));
if( zk == NULL )
{
    printf( "no enough memory for array zk\n" );
    return -1;
}

xx = ( double * )malloc((size_t)( sizeof(double) * nnx ));
if( xx == NULL )
{
    printf( "no enough memory for array xx\n" );
    return -1;
}

yy = ( double * )malloc((size_t)( sizeof(double) * nny ));
if( yy == NULL )
{
    printf( "no enough memory for array yy\n" );
    return -1;
}

zz = ( double * )malloc((size_t)( sizeof(double) * nnz ));
if( zz == NULL )
{
    printf( "no enough memory for array zz\n" );
    return -1;
}
```

```
s = ( double * )malloc((size_t)( sizeof(double) * nnx*nny*nnz ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double) *
(nxd*nxd*nyd*nyd*nzd*nzd+nxd*nyd*nzd
 +mx*(nxd+2)+my*(nyd+2)+2*mz+3) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tnxd = %6d mx = %6d nnx = %6d\n",
 nxd, mx, nnx );
printf( "\tnyd = %6d my = %6d nny = %6d\n",
 nyd, my, nny );
printf( "\tnzd = %6d mz = %6d nnz = %6d\n",
 nzd, mz, nnz );

for( i=0 ; i<nxd ; i++ )
{
    xd[i] = 0.1 * (double)i;
}
for( j=0 ; j<nyd ; j++ )
{
    yd[j] = 0.1 * (double)j;
}
for( k=0 ; k<nzd ; k++ )
{
    zd[k] = 0.1 * (double)k;
}

for( k=0 ; k<nzd ; k++ )
{
for( j=0 ; j<nyd ; j++ )
{
for( i=0 ; i<nxd ; i++ )
{
    fd[i+j*nxd+k*nxd*nyd] = 10.0
    +1.0/(0.03+(xd[i]-0.5)*(xd[i]-0.5))
    +1.0/(0.04+(yd[j]-0.2)*(yd[j]-0.2))
    +1.0/(0.05+(zd[k]-0.6)*(zd[k]-0.6));
}
}
}

for( i=0 ; i<nxd+mx ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
}
for( j=0 ; j<nyd+my ; j++ )
{
    fscanf( fp, "%lf", &yk[j] );
}
for( k=0 ; k<nzd+mz ; k++ )
{
    fscanf( fp, "%lf", &zk[k] );
}

for( i=0 ; i<nnx ; i++ )
{
    xx[i] = 0.1 * (double)i;
}
for( j=0 ; j<nny ; j++ )
{
    yy[j] = 0.1 * (double)j;
}
for( k=0 ; k<nnz ; k++ )
{
    zz[k] = 0.1 * (double)k;
}

fclose( fp );

ierr = ASL_dgisi3(xd, nxd, yd, nyd, zd, nzd, fd, xk, mx, yk, my, zk,
     mz, xx, nnx, yy, nny, zz, nnz, s, wk, iwk);

printf( "\n     ** Output **\n" );
printf( "\n\tierr = %6d\n", ierr );

if( ierr < 3000 )
{
    sumt2 = 0.0;
    sumd2 = 0.0;
    for( i=0 ; i<nnx ; i++ )
    {
```

```
            for( j=0 ; j<nny ; j++ )
            {
            for( k=0 ; k<nnz ; k++ )
            {
                ff = 10.0
                   +1.0/(0.03+(xx[i]-0.5)*(xx[i]-0.5))
                   +1.0/(0.04+(yy[j]-0.2)*(yy[j]-0.2))
                   +1.0/(0.05+(zz[k]-0.6)*(zz[k]-0.6));
                sumt2 += ff * ff;
                sumd2 += (ff-s[i+j*nnx+k*nnx*nny])
                       * (ff-s[i+j*nnx+k*nnx*nny]);
            }
            }
            }
            sumt2 = sqrt(sumt2);
            sumd2 = sqrt(sumd2);
            fit = 100.0;
            if( ( sumt2 != 0.0 )||( sumd2 != 0.0 ) )
            {
                fit = ( sumt2 / ( sumt2 + sumd2 ) ) * 100.0;
            }
            printf( "\n\tFitting rate = %8.3g\n", fit );
        }

        free( iwk );
        free( xd );
        free( yd );
        free( zd );
        free( fd );
        free( xk );
        free( yk );
        free( zk );
        free( xx );
        free( yy );
        free( zz );
        free( s );
        free( wk );

        return 0;
    }
```

(d) Output results

```
*** ASL_dgisi3 ***

 ** Input **

nxd =      9 mx =      4 nnx =      9
nyd =      9 my =      4 nny =      9
nzd =      9 mz =      4 nnz =      9

 ** Output **

ierr =      0

Fitting rate =      100
```

6.5.5 ASL_dgiss1, ASL_rgiss1
B-Spline Smoothing (One-Dimensional Data)

(1) **Function**

ASL_dgiss1 or ASL_rgiss1 obtains the following spline function of degree $(m-1)$ for smoothing the data $f_i \ \ (i = 1, 2, \cdots, N)$ at the sample points $x_i \ \ (i = 1, 2, \cdots, N)$:

$$S(x) \quad = \quad \sum_{i=1}^{n+m} c_i N_{mi}(x)$$

and calculates the approximation function values at the specified points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiss1  (xd, nd, fd, xk, m, xx, nn, s, rs, &aic, wk);
```

Single precision:

```
ierr = ASL_rgiss1  (xd, nd, fd, xk, m, xx, nn, s, rs, &aic, wk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{D∗ / R∗}	nd	Input	Sample points x_i.
2	nd	I	1	Input	Number of sample points N.
3	fd	{D∗ / R∗}	nd	Input	Data f_i given at sample points x_i.
4	xk	{D∗ / R∗}	nd + m	Input	Knots (including additional knots) ξ_i.
5	m	I	1	Input	B–spline order m.
6	xx	{D∗ / R∗}	nn	Input	x coordinates for calculating approximation function values.
7	nn	I	1	Input	Number of points for calculating approximation function values.
8	s	{D∗ / R∗}	nn	Output	Approximation function value $S(x)$ at $x =$ xx[i − 1] (i = 1, 2, ⋯, nn).
9	rs	{D∗ / R∗}	nd	Output	Residual $S(x_i) - f_i$.

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	aic	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	1	Output	Akaike's Information Criterion AIC.
11	wk	$\left\{\begin{matrix}\text{D}* \\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area. **Size**:$\text{nd}^2 + \text{nd} + 2 \times \text{m} + 1$
12	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{nd} \geq 1$

(b) $\text{nd} - \text{m} \geq 1, \text{m} \geq 1$

(c) $\text{nn} \geq 1$

(d) $\text{xd}[0] < \text{xd}[1] < \cdots < \text{xd}[\text{nd} - 1]$

(e) $\text{xk}[\text{m} - 1] \leq \text{xd}[\text{i} - 1] \leq \text{xk}[\text{nd}] \quad (\text{i} = 1, 2, \cdots, \text{nd})$

(f) $\text{xk}[0] \leq \text{xk}[1] \leq \cdots \leq \text{xk}[\text{nd} + \text{m} - 1]$

(g) $\text{xk}[\text{i} - 1] < \text{xk}[\text{i} + \text{m} - 1] \quad (\text{i} = 1, 2, \cdots, \text{nd})$

(h) The sample values $\text{xd}[\text{i} - 1] \quad (\text{i} = 1, 2, \cdots, \text{nd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(i) $\text{xk}[\text{m} - 1] \leq \text{xx}[\text{i} - 1] \leq \text{xk}[\text{nd}] \quad (\text{i} = 1, 2, \cdots, \text{nn})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
2100	When the normalized equation is solved, there existed the diagonal element which was close to zero in the LU decomposition. The precise solutions may not be obtained.	Processing continues.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3400	Restriction (d) was not satisfied.	
3500	Restriction (e) was not satisfied.	
3600	Restriction (f) was not satisfied.	
3700	Restriction (g) was not satisfied.	
3800	Restriction (h) was not satisfied.	
3900	Restriction (i) was not satisfied.	
4000	The normalized equation could not be solved.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(b) Number of internal knots is nd − m.

(7) **Example**

(a) Problem

Perform the fitting to the following data f_i using a cubic spline function.

$$f_i = \frac{1}{0.01 + (x_i - 0.3)^2} + e_i \quad (x_i = 0.0, 0.1, \cdots, 1.0)$$

Here, e_i are mutually independent errors that obey a normal distribution having mean 0 and variance 1.0.

(b) Input data

Sample points (xd, fd), nd=11, knots xk, m=4, x coordinates for calculating interpolation values xx and nn=19.

(c) Main Program

```
/*      C interface example for ASL_dgiss1 */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *xd;
    int nd;
    double *fd;
    double *xk;
    int m;
    double *xx;
    int nn;
    double *s;
    double *rs;
    double aic;
    double *wk;
    double am,sg;
    int ierr;
    int i,ix,iy;
    double *e;
    FILE *fp;

    fp = fopen( "dgiss1.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgiss1 ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &nd );
    fscanf( fp, "%d", &m );
    fscanf( fp, "%d", &nn );

    xd = ( double * )malloc((size_t)( sizeof(double) * nd ));
    if( xd == NULL )
    {
        printf( "no enough memory for array xd\n" );
        return -1;
    }

    fd = ( double * )malloc((size_t)( sizeof(double) * nd ));
    if( fd == NULL )
    {
        printf( "no enough memory for array fd\n" );
        return -1;
    }

    xk = ( double * )malloc((size_t)( sizeof(double) * (nd+m) ));
    if( xk == NULL )
    {
        printf( "no enough memory for array xk\n" );
        return -1;
    }
```

```
xx = ( double * )malloc((size_t)( sizeof(double) * nn ));
if( xx == NULL )
{
    printf( "no enough memory for array xx\n" );
    return -1;
}

s = ( double * )malloc((size_t)( sizeof(double) * nn ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

rs = ( double * )malloc((size_t)( sizeof(double) * nd ));
if( rs == NULL )
{
    printf( "no enough memory for array rs\n" );
    return -1;
}

e = ( double * )malloc((size_t)( sizeof(double) * nd ));
if( e == NULL )
{
    printf( "no enough memory for array e\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double) *
(nd*nd+nd+2*m+1) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tnd = %6d m = %6d nn = %6d\n", nd,m,nn );

printf( "\n\n\txd\n\n" );
for( i=0 ; i<nd ; i++ )
{
    xd[i] = 0.1 * (double)i;
    printf( "\txd[%6d] = %8.3g\n", i, xd[i] );
}

ix=1;
iy=1;
am = 0.0;
sg = 1.0;

ierr = ASL_djdbno(nd, am, sg, &ix, &iy, e);

for( i=0 ; i<nd ; i++ )
{
    fd[i] = 1.0/(0.01+(xd[i]-0.3)*(xd[i]-0.3))+e[i];
}

printf( "\n\n\txk\n\n" );
for( i=0 ; i<nd+m ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
    printf( "\txk[%6d] = %8.3g\n", i, xk[i] );
}

for( i=0 ; i<nn ; i++ )
{
    xx[i] = 0.05 * (i+1);
}

fclose( fp );

ierr = ASL_dgiss1(xd, nd, fd, xk, m, xx, nn, s, rs, &aic, wk);

printf( "\n     ** Output **\n\n" );
printf( "\n\n\tierr = %6d\n", ierr );
printf( "\n\n\taic  = %8.3g\n", aic );
printf( "\n\n\tValues of approximating spline function on sample points\n\n");
printf( "\t       xx  \t       s\n\n" );
for( i=0 ; i<nn ; i++ )
{
    printf( "\t%8.3g\t%8.3g\n", xx[i], s[i] );
}

free( xd );
free( fd );
free( xk );
free( xx );
free( s );
free( rs );
free( e );
free( wk );
```

```
    return 0;
}
```

(d) Output results

```
*** ASL_dgiss1 ***

 ** Input **

nd =     11 m =      4 nn =     19


xd

xd[     0] =        0
xd[     1] =      0.1
xd[     2] =      0.2
xd[     3] =      0.3
xd[     4] =      0.4
xd[     5] =      0.5
xd[     6] =      0.6
xd[     7] =      0.7
xd[     8] =      0.8
xd[     9] =      0.9
xd[    10] =        1


xk

xk[     0] =        0
xk[     1] =        0
xk[     2] =        0
xk[     3] =        0
xk[     4] =      0.2
xk[     5] =      0.3
xk[     6] =      0.4
xk[     7] =      0.5
xk[     8] =      0.6
xk[     9] =      0.7
xk[    10] =      0.8
xk[    11] =        1
xk[    12] =        1
xk[    13] =        1
xk[    14] =        1

 ** Output **


ierr =      0


aic  =     -677


Values of approximating spline function on sample points

       xx          s

     0.05        14.9
      0.1        20.3
     0.15        30.4
      0.2        51.4
     0.25          83
      0.3        99.9
     0.35        81.1
      0.4        48.9
     0.45        29.3
      0.5        20.2
     0.55        14.4
      0.6        10.4
     0.65        8.09
      0.7        6.26
     0.75        3.91
      0.8        1.97
     0.85        1.51
      0.9        1.97
     0.95        2.42
```

6.5.6 ASL_dgiss2, ASL_rgiss2
B-Spline Smoothing (Two-Dimensional Data)

(1) **Function**

ASL_dgiss2 or ASL_rgiss2 obtains the following spline function for smoothing the data f_{ij} $(i = 1, 2, \cdots, N_x; j = 1, 2, \cdots, N_y)$ at the sample points (x_i, y_j) $(i = 1, 2, \cdots, N_x; j = 1, 2, \cdots, N_y)$:

$$S(x, y) = \sum_{i=1}^{h+m} \sum_{j=1}^{k+n} c_{ij} N_{mi}(x) N_{nj}(y)$$

and calculates the approximate function values at the specified grid points.

(2) **Usage**

Double precision:

ierr = ASL_dgiss2 (xd, nxd, yd, nyd, fd, xk, mx, yk, my, xx, nnx, yy, nny, s, rs, &aic, wk, iwk);

Single precision:

ierr = ASL_rgiss2 (xd, nxd, yd, nyd, fd, xk, mx, yk, my, xx, nnx, yy, nny, s, rs, &aic, wk, iwk);

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex I: { int as for 32bit Integer / long as for 64bit Integer }
R:Single precision real C:Single precision complex

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{D∗ / R∗}	nxd	Input	Sample points in x direction x_i.
2	nxd	I	1	Input	Number of sample points in x direction N_x.
3	yd	{D∗ / R∗}	nyd	Input	Sample points in y direction y_j.
4	nyd	I	1	Input	Number of sample points in y direction N_y.
5	fd	{D∗ / R∗}	nxd×nyd	Input	Data f_{ij} given at sample points (x_i, y_j).
6	xk	{D∗ / R∗}	nxd + mx	Input	Knots (including additional knots) in x direction ξ_i.
7	mx	I	1	Input	x–direction B–spline order m.
8	yk	{D∗ / R∗}	nyd + my	Input	Knots (including additional knots) in y direction ζ_j.
9	my	I	1	Input	y–direction B–spline order n.

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
10	xx	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nnx	Input	x coordinates for calculating approximate function values.
11	nnx	I	1	Input	Number of x–direction points for calculating approximate function values.
12	yy	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nny	Input	y coordinates for calculating approximate function values.
13	nny	I	1	Input	Number of y–direction points for calculating approximate function values.
14	s	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nnx×nny	Output	Approximation function value $S(x, y)$ at $(x, y) = (\text{xx[i} - 1], \text{yy[j} - 1])$ $(\text{i} = 1, 2, \cdots, \text{nnx}\ ;\ \text{j} = 1, 2, \cdots, \text{nny})$.
15	rs	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nxd×nyd	Output	Residual $S(x_i, y_j) - f_{ij}$.
16	aic	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	1	Output	Akaike's Information Criterion AIC.
17	wk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area. **Size**: $\text{nxd}^2 \times (\text{nyd}^2 + 1) + \text{nxd} \times (\text{nyd} + \text{mx}) + 2 \times \text{mx} + 2 \times \text{my} + 2$
18	iwk	I*	nxd + nyd	Work	Work area.
19	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\text{nxd} \geq 1, \text{nyd} \geq 1$

(b) $\text{nxd} - \text{mx} \geq 1, \text{nyd} - \text{my} \geq 1, \text{mx} \geq 1, \text{my} \geq 1$

(c) $\text{nnx} \geq 1, \text{nny} \geq 1$

(d) $\text{xd}[0] < \text{xd}[1] < \cdots < \text{xd}[\text{nxd} - 1]$

(e) $\text{yd}[0] < \text{yd}[1] < \cdots < \text{yd}[\text{nyd} - 1]$

(f) $\text{xk}[\text{mx} - 1] \leq \text{xd}[\text{i} - 1] \leq \text{xk}[\text{nxd}]\ \ (\text{i} = 1, 2, \cdots, \text{nxd})$

(g) $\text{yk}[\text{my} - 1] \leq \text{yd}[\text{j} - 1] \leq \text{yk}[\text{nyd}]\ \ (\text{j} = 1, 2, \cdots, \text{nyd})$

(h) $\text{xk}[0] \leq \text{xk}[1] \leq \cdots \leq \text{xk}[\text{nxd} + \text{mx} - 1]$

(i) $\text{yk}[0] \leq \text{yk}[1] \leq \cdots \leq \text{yk}[\text{nyd} + \text{my} - 1]$

(j) $\text{xk}[\text{i} - 1] < \text{xk}[\text{i} + \text{mx} - 1]\ \ (\text{i} = 1, 2, \cdots, \text{nxd})$

(k) $\text{yk}[\text{j} - 1] < \text{yk}[\text{j} + \text{my} - 1]\ \ (\text{j} = 1, 2, \cdots, \text{nyd})$

(l) The sample values $\text{xd}[\text{i} - 1]\ \ (\text{i} = 1, 2, \cdots, \text{nxd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(m) The sample values $\text{yd}[\text{j} - 1]\ \ (\text{j} = 1, 2, \cdots, \text{nyd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(n) $\text{xk}[\text{mx} - 1] \leq \text{xx}[\text{i} - 1] \leq \text{xk}[\text{nxd}]\ \ (\text{i} = 1, 2, \cdots, \text{nnx})$

(o) $\text{yk}[\text{my} - 1] \leq \text{yy}[\text{j} - 1] \leq \text{yk}[\text{nyd}]\ \ (\text{j} = 1, 2, \cdots, \text{nny})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
2100	When the normalized equation is solved, there existed the diagonal element which was close to zero in the LU decomposition. The precise solutions may not be obtained.	Processing continues.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3400	Restriction (d) was not satisfied.	
3410	Restriction (e) was not satisfied.	
3500	Restriction (f) was not satisfied.	
3510	Restriction (g) was not satisfied.	
3600	Restriction (h) was not satisfied.	
3610	Restriction (i) was not satisfied.	
3700	Restriction (j) was not satisfied.	
3710	Restriction (k) was not satisfied.	
3800	Restriction (l) was not satisfied.	
3810	Restriction (m) was not satisfied.	
3900	Restriction (n) was not satisfied.	
3910	Restriction (o) was not satisfied.	
4000	The normalized equation could not be solved.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(b) Number of internal knots is $(\mathrm{nxd}-\mathrm{mx}, \mathrm{nyd}-\mathrm{my})$.

(7) **Example**

(a) Problem

Perform the fitting to the following data f_{ij} using a cubic spline function.

$$f_{ij} = \frac{1}{0.01+(x_i-0.3)^2} + \frac{1}{0.02+(y_j-0.4)^2} + e_{ij}$$
$$(x_i = 0.0, 0.1, \cdots, 1.0; y_j = 0.0, 0.1, \cdots, 1.0)$$

Here, e_{ij} are mutually independent errors that obey a normal distribution having mean 0 and variance 1.0.

(b) Input data

Sample points (xd, yd, fd), nxd=11, nyd=11,
knot xk, mx=4, knot yk, my=4,
x coordinates for calculating interpolation values xx, nnx=11,
y coordinates for calculating interpolation values yy and nny=11.

(c) Main Program

```
/*      C interface example for ASL_dgiss2 */

#include <stdio.h>
#include <stdlib.h>
#include <asl.h>

int main()
{
    double *xd;
    int nxd;
    double *yd;
    int nyd;
    double *fd;
    double *xk;
    int mx;
    double *yk;
    int my;
    double *xx;
    int nnx;
    double *yy;
    int nny;
    double *s;
    double *rs;
    double aic;
    double *wk;
    int *iwk;
    int ierr;
    int i,j,ix,iy;
    double *e,am,sg;
    FILE *fp;

    fp = fopen( "dgiss2.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgiss2 ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &nxd );
    fscanf( fp, "%d", &mx );
    fscanf( fp, "%d", &nnx );
    fscanf( fp, "%d", &nyd );
    fscanf( fp, "%d", &my );
    fscanf( fp, "%d", &nny );


    iwk = ( int * )malloc((size_t)( sizeof(int) * (nxd+nyd) ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    xd = ( double * )malloc((size_t)( sizeof(double) * nxd ));
    if( xd == NULL )
    {
        printf( "no enough memory for array xd\n" );
        return -1;
    }

    yd = ( double * )malloc((size_t)( sizeof(double) * nyd ));
    if( yd == NULL )
    {
        printf( "no enough memory for array yd\n" );
        return -1;
    }

    fd = ( double * )malloc((size_t)( sizeof(double) * nxd*nyd ));
    if( fd == NULL )
    {
        printf( "no enough memory for array fd\n" );
        return -1;
    }

    xk = ( double * )malloc((size_t)( sizeof(double) * (nxd+mx) ));
    if( xk == NULL )
    {
        printf( "no enough memory for array xk\n" );
        return -1;
    }

    yk = ( double * )malloc((size_t)( sizeof(double) * (nyd+my) ));
    if( yk == NULL )
    {
        printf( "no enough memory for array yk\n" );
        return -1;
```

```
}

xx = ( double * )malloc((size_t)( sizeof(double) * nnx ));
if( xx == NULL )
{
    printf( "no enough memory for array xx\n" );
    return -1;
}

yy = ( double * )malloc((size_t)( sizeof(double) * nny ));
if( yy == NULL )
{
    printf( "no enough memory for array yy\n" );
    return -1;
}

s = ( double * )malloc((size_t)( sizeof(double) * nnx * nny ));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

rs = ( double * )malloc((size_t)( sizeof(double)*nxd*nyd ));
if( rs == NULL )
{
    printf( "no enough memory for array rs\n" );
    return -1;
}

e = ( double * )malloc((size_t)( sizeof(double)*nxd*nyd ));
if( e == NULL )
{
    printf( "no enough memory for array e\n" );
    return -1;
}

wk = ( double * )malloc((size_t)( sizeof(double)*
(nxd*nxd*(nyd*nyd+1)+nxd*(nyd+mx)+mx*mx+my*my+2) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tnxd = %6d mx = %6d nnx = %6d\n",
    nxd, mx, nnx );
printf( "\tnyd = %6d my = %6d nny = %6d\n",
    nyd, my, nny );

printf( "\n\n\txd\n\n" );
for( i=0 ; i<nxd ; i++ )
{
    xd[i] = 0.1 * (double)i;
    printf( "\txd[%6d] = %8.3g\n", i, xd[i] );
}

printf( "\n\n\tyd\n\n" );
for( j=0 ; j<nyd ; j++ )
{
    yd[j] = 0.1 * (double)j;
    printf( "\tyd[%6d] = %8.3g\n", j, yd[j] );
}

ix=1;
iy=1;
am = 0.0;
sg = 1.0;

ierr = ASL_djdbno(nxd*nyd, am, sg, &ix, &iy, e);

for( j=0 ; j<nyd ; j++ )
{
    for( i=0 ; i<nxd ; i++ )
    {
        fd[i+nxd*j]=
         1.0/(0.01+(xd[i]-0.3)*(xd[i]-0.3))
        +1.0/(0.02+(yd[j]-0.4)*(yd[j]-0.4))
        +e[j*nxd+i];
    }
}

printf( "\n\txk\n\n" );
for( i=0 ; i<nxd+mx ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
    printf( "\txk[%6d] = %8.3g\n", i, xk[i] );
}

printf( "\n\tyk\n\n" );
```

```
    for( j=0 ; j<nyd+my ; j++ )
    {
        fscanf( fp, "%lf", &yk[j] );
        printf( "\tyk[%6d] = %8.3g\n", j, yk[j] );
    }

    for( i=0 ; i<nnx ; i++ )
    {
        xx[i] = 0.1 * (double)i;
    }

    for( j=0 ; j<nny ; j++ )
    {
        yy[j] = 0.1 * (double)j;
    }

    fclose( fp );

    ierr = ASL_dgiss2(xd, nxd, yd, nyd, fd, xk, mx, yk, my, xx, nnx,
         yy, nny, s, rs, &aic, wk, iwk);

    printf( "\n     ** Output **\n\n" );
    printf( "\n\n\tierr = %6d\n", ierr );
    printf( "\n\n\taic  = %8.3g\n", aic );
    printf( "\n\n\tValues of an approximating spline function on sample points\n\n" );
    printf( "\n\n\t      xx   \t      yy   \t     s\n\n" );
    for( i=0 ; i<nnx ; i++ )
    {
        for( j=0 ; j<nny ; j++ )
        {
            printf( "\t%8.3g\t%8.3g\t%8.3g\n", xx[i], yy[j], s[i+nnx*j] );
        }
    }

    free( iwk );
    free( xd );
    free( yd );
    free( fd );
    free( xk );
    free( yk );
    free( xx );
    free( yy );
    free( s );
    free( rs );
    free( e );
    free( wk );

    return 0;
}
```

(d) Output results

```
*** ASL_dgiss2 ***

 ** Input **

nxd =     11 mx =      4 nnx =     11
nyd =     11 my =      4 nny =     11


xd

xd[     0] =        0
xd[     1] =      0.1
xd[     2] =      0.2
xd[     3] =      0.3
xd[     4] =      0.4
xd[     5] =      0.5
xd[     6] =      0.6
xd[     7] =      0.7
xd[     8] =      0.8
xd[     9] =      0.9
xd[    10] =        1


yd

yd[     0] =        0
yd[     1] =      0.1
yd[     2] =      0.2
yd[     3] =      0.3
yd[     4] =      0.4
yd[     5] =      0.5
yd[     6] =      0.6
yd[     7] =      0.7
yd[     8] =      0.8
yd[     9] =      0.9
yd[    10] =        1
```

```
xk

xk[     0] =        0
xk[     1] =        0
xk[     2] =        0
xk[     3] =        0
xk[     4] =      0.2
xk[     5] =      0.3
xk[     6] =      0.4
xk[     7] =      0.5
xk[     8] =      0.6
xk[     9] =      0.7
xk[    10] =      0.8
xk[    11] =        1
xk[    12] =        1
xk[    13] =        1
xk[    14] =        1

yk

yk[     0] =        0
yk[     1] =        0
yk[     2] =        0
yk[     3] =        0
yk[     4] =      0.2
yk[     5] =      0.3
yk[     6] =      0.4
yk[     7] =      0.5
yk[     8] =      0.6
yk[     9] =      0.7
yk[    10] =      0.8
yk[    11] =        1
yk[    12] =        1
yk[    13] =        1
yk[    14] =        1

 ** Output **


ierr =      0

aic  = -6.7e+03

Values of an approximating spline function on sample points


      xx          yy         s

       0           0        13.6
       0         0.1        17.9
       0         0.2        24.9
       0         0.3        44.2
       0         0.4        60.6
       0         0.5        43.4
       0         0.6        26.1
       0         0.7        18.4
       0         0.8        15.7
       0         0.9        15.4
       0           1          13
     0.1           0        25.8
     0.1         0.1        29.5
     0.1         0.2          34
     0.1         0.3        52.7
     0.1         0.4          72
     0.1         0.5        53.3
     0.1         0.6        35.1
     0.1         0.7        28.8
     0.1         0.8        25.9
     0.1         0.9        24.2
     0.1           1        21.5
     0.2           0        56.9
     0.2         0.1        60.6
     0.2         0.2        66.2
     0.2         0.3          84
     0.2         0.4         100
     0.2         0.5        84.9
     0.2         0.6        64.9
     0.2         0.7        59.5
     0.2         0.8        55.1
     0.2         0.9        53.5
     0.2           1        51.6
     0.3           0         105
     0.3         0.1         110
     0.3         0.2         117
     0.3         0.3         134
     0.3         0.4         150
     0.3         0.5         134
     0.3         0.6         115
```

0.3	0.7	107
0.3	0.8	108
0.3	0.9	103
0.3	1	104
0.4	0	54.5
0.4	0.1	60
0.4	0.2	67.3
0.4	0.3	83.5
0.4	0.4	99.8
0.4	0.5	83.2
0.4	0.6	64.6
0.4	0.7	58.4
0.4	0.8	56.3
0.4	0.9	53.9
0.4	1	54
0.5	0	25.8
0.5	0.1	28.1
0.5	0.2	39.1
0.5	0.3	53.3
0.5	0.4	71.3
0.5	0.5	52.5
0.5	0.6	35.8
0.5	0.7	28
0.5	0.8	25.3
0.5	0.9	23.1
0.5	1	21.7
0.6	0	15.9
0.6	0.1	20.6
0.6	0.2	26.5
0.6	0.3	42.6
0.6	0.4	59.9
0.6	0.5	43.2
0.6	0.6	26.3
0.6	0.7	17.6
0.6	0.8	16.7
0.6	0.9	12.7
0.6	1	11.8
0.7	0	11.8
0.7	0.1	14.4
0.7	0.2	21.3
0.7	0.3	39.9
0.7	0.4	55.4
0.7	0.5	38.7
0.7	0.6	21.9
0.7	0.7	15.2
0.7	0.8	11.7
0.7	0.9	8.95
0.7	1	9.04
0.8	0	7.53
0.8	0.1	13.6
0.8	0.2	21.6
0.8	0.3	35.7
0.8	0.4	53.6
0.8	0.5	36.9
0.8	0.6	21.5
0.8	0.7	11.6
0.8	0.8	8.22
0.8	0.9	8.12
0.8	1	6.2
0.9	0	7.53
0.9	0.1	13
0.9	0.2	17.9
0.9	0.3	35.8
0.9	0.4	52.5
0.9	0.5	35.1
0.9	0.6	19.2
0.9	0.7	11.2
0.9	0.8	8.4
0.9	0.9	4.64
0.9	1	4.93
1	0	7.45
1	0.1	9.82
1	0.2	18.4
1	0.3	36.2
1	0.4	52.4
1	0.5	35.8
1	0.6	18.2
1	0.7	12.2
1	0.8	7.59
1	0.9	4.93
1	1	5.85

6.5.7 ASL_dgiss3, ASL_rgiss3
B-Spline Smoothing (Three-Dimensional Data)

(1) **Function**

ASL_dgiss3 or ASL_rgiss3 obtains the following spline function for smoothing the data f_{ijk} $(i = 1, 2, \cdots, N_x;$ $j = 1, 2, \cdots, N_y;$ $k = 1, 2, \cdots, N_z)$ at the sample points (x_i, y_j, z_k) $(i = 1, 2, \cdots, N_x; j = 1, 2, \cdots, N_y; k = 1, 2, \cdots, N_z)$:

$$S(x, y, z) = \sum_{i=1}^{n_x+m_x} \sum_{j=1}^{n_y+m_y} \sum_{k=1}^{n_z+m_z} c_{ijk} N_{m_x i}(x) N_{m_y j}(y) N_{m_z k}(z)$$

and calculates the approximate function values at the specified grid points.

(2) **Usage**

Double precision:

```
ierr = ASL_dgiss3 (xd, nxd, yd, nyd, zd, nzd, fd, xk, mx, yk, my, zk, mz, xx, nnx, yy, nny, zz,
                   nnz, s, rs, &aic, wk, iwk);
```

Single precision:

```
ierr = ASL_rgiss3 (xd, nxd, yd, nyd, zd, nzd, fd, xk, mx, yk, my, zk, mz, xx, nnx, yy, nny, zz,
                   nnz, s, rs, &aic, wk, iwk);
```

(3) **Arguments and Return Value**

D:Double precision real Z:Double precision complex
R:Single precision real C:Single precision complex
I: { int as for 32bit Integer / long as for 64bit Integer }

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
1	xd	{D∗ / R∗}	nxd	Input	Sample points in x direction x_i.
2	nxd	I	1	Input	Number of sample points in x direction N_x.
3	yd	{D∗ / R∗}	nyd	Input	Sample points in y direction y_j.
4	nyd	I	1	Input	Number of sample points in y direction N_y.
5	zd	{D∗ / R∗}	nzd	Input	Sample points in z direction z_k.
6	nzd	I	1	Input	Number of sample points in z direction N_z.
7	fd	{D∗ / R∗}	See Contents	Input	Data f_{ijk} given at sample points (x_i, y_j, z_k). **Size**: nxd×nyd×nzd

No.	Argument and Return Value	Type	Size	Input/ Output	Contents
8	xk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nxd + mx	Input	Knots (including additional knots) in x direction ξ_i.
9	mx	I	1	Input	x–direction B–spline order m_x.
10	yk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nyd + my	Input	Knots (including additional knots) in y direction ζ_j.
11	my	I	1	Input	y–direction B–spline order m_y.
12	zk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nzd + mz	Input	Knots (including additional knots) in z direction η_k.
13	mz	I	1	Input	z–direction B–spline order m_z.
14	xx	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nnx	Input	x coordinates for calculating approximate function values.
15	nnx	I	1	Input	Number of x–direction points for calculating approximate function values.
16	yy	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nny	Input	y coordinates for calculating approximate function values.
17	nny	I	1	Input	Number of y–direction points for calculating approximate function values.
18	zz	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	nnz	Input	z coordinates for calculating approximate function values.
19	nnz	I	1	Input	Number of z–direction points for calculating approximate function values.
20	s	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Output	Approximation function value $S(x, y, z)$ at $(x, y) = (\text{xx[i} - 1], \text{yy[j} - 1], \text{zz[k} - 1])$ $(\text{i} = 1, 2, \cdots, \text{nnx}\ ;\ \text{j} = 1, 2, \cdots, \text{nny}\ ;$ $\text{k} = 1, 2, \cdots, \text{nnz})$. **Size**: nnx×nny×nnz
21	rs	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Output	Residual $S(x_i, y_j, z_k) - f_{ijk}$. **Size**: nxd×nyd×nzd
22	aic	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	1	Output	Akaike's Information Criterion AIC.
23	wk	$\left\{\begin{matrix}\text{D}*\\ \text{R}*\end{matrix}\right\}$	See Contents	Work	Work area. **Size**: $\text{nxd}^2 \times \text{nyd}^2 \times \text{nzd}^2 + \text{nxd} \times \text{nyd} \times \text{nzd} + \text{nxd}^2 \times \text{nyd}^2 + \text{nxd}^2 + \text{mx} \times (\text{nxd} + 2) + \text{my} \times (\text{nyd} + 2) + 2 \times \text{mz} + 3$
24	iwk	I*	See Contents	Work	Work area. **Size**: nxd × nyd × nzd + nxd + nyd + nzd
25	ierr	I	1	Output	Error indicator (Return Value)

(4) **Restrictions**

(a) $\mathrm{nxd} \geq 1, \mathrm{nyd} \geq 1, \mathrm{nzd} \geq 1$

(b) $\mathrm{nxd} - \mathrm{mx} \geq 1, \mathrm{nyd} - \mathrm{my} \geq 1, \mathrm{nzd} - \mathrm{mz} \geq 1, \mathrm{mx} \geq 1, \mathrm{my} \geq 1, \mathrm{mz} \geq 1$

(c) $\mathrm{nnx} \geq 1, \mathrm{nny} \geq 1, \mathrm{nnz} \geq 1$

(d) $\mathrm{xd}[0] < \mathrm{xd}[1] < \cdots < \mathrm{xd}[\mathrm{nxd} - 1]$

(e) $\mathrm{yd}[0] < \mathrm{yd}[1] < \cdots < \mathrm{yd}[\mathrm{nyd} - 1]$

(f) $\mathrm{zd}[0] < \mathrm{zd}[1] < \cdots < \mathrm{zd}[\mathrm{nzd} - 1]$

(g) $\mathrm{xk}[\mathrm{mx} - 1] \leq \mathrm{xd}[\mathrm{i} - 1] \leq \mathrm{xk}[\mathrm{nxd}] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nxd})$

(h) $\mathrm{yk}[\mathrm{my} - 1] \leq \mathrm{yd}[\mathrm{j} - 1] \leq \mathrm{yk}[\mathrm{nyd}] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nyd})$

(i) $\mathrm{zk}[\mathrm{mz} - 1] \leq \mathrm{zd}[\mathrm{k} - 1] \leq \mathrm{zk}[\mathrm{nzd}] \quad (\mathrm{k} = 1, 2, \cdots, \mathrm{nzd})$

(j) $\mathrm{xk}[0] \leq \mathrm{xk}[1] \leq \cdots \leq \mathrm{xk}[\mathrm{nxd} + \mathrm{mx} - 1]$

(k) $\mathrm{yk}[0] \leq \mathrm{yk}[1] \leq \cdots \leq \mathrm{yk}[\mathrm{nyd} + \mathrm{my} - 1]$

(l) $\mathrm{zk}[0] \leq \mathrm{zk}[1] \leq \cdots \leq \mathrm{zk}[\mathrm{nzd} + \mathrm{mz} - 1]$

(m) $\mathrm{xk}[\mathrm{i} - 1] < \mathrm{xk}[\mathrm{i} + \mathrm{mx} - 1] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nxd})$

(n) $\mathrm{yk}[\mathrm{j} - 1] < \mathrm{yk}[\mathrm{j} + \mathrm{my} - 1] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nyd})$

(o) $\mathrm{zk}[\mathrm{k} - 1] < \mathrm{zk}[\mathrm{k} + \mathrm{mz} - 1] \quad (\mathrm{k} = 1, 2, \cdots, \mathrm{nzd})$

(p) The sample values $\mathrm{xd}[\mathrm{i} - 1] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nxd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(q) The sample values $\mathrm{yd}[\mathrm{j} - 1] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nyd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(r) The sample values $\mathrm{zd}[\mathrm{k} - 1] \quad (\mathrm{k} = 1, 2, \cdots, \mathrm{nzd})$ satisfy the Schoenberg–Whitney conditions (see Section 6.1.2).

(s) $\mathrm{xk}[\mathrm{mx} - 1] \leq \mathrm{xx}[\mathrm{i} - 1] \leq \mathrm{xk}[\mathrm{nxd}] \quad (\mathrm{i} = 1, 2, \cdots, \mathrm{nnx})$

(t) $\mathrm{yk}[\mathrm{my} - 1] \leq \mathrm{yy}[\mathrm{j} - 1] \leq \mathrm{yk}[\mathrm{nyd}] \quad (\mathrm{j} = 1, 2, \cdots, \mathrm{nny})$

(u) $\mathrm{zk}[\mathrm{mz} - 1] \leq \mathrm{zz}[\mathrm{k} - 1] \leq \mathrm{zk}[\mathrm{nzd}] \quad (\mathrm{k} = 1, 2, \cdots, \mathrm{nnz})$

(5) **Error indicator (Return Value)**

ierr value	Meaning	Processing
0	Normal termination.	
2100	When the normalized equation is solved, there existed the diagonal element which was close to zero in the LU decomposition. The precise solutions may not be obtained.	Processing continues.
3000	Restriction (a) was not satisfied.	Processing is aborted.
3100	Restriction (b) was not satisfied.	
3200	Restriction (c) was not satisfied.	
3400	Restriction (d) was not satisfied.	
3410	Restriction (e) was not satisfied.	
3420	Restriction (f) was not satisfied.	
3500	Restriction (g) was not satisfied.	
3510	Restriction (h) was not satisfied.	
3520	Restriction (i) was not satisfied.	
3600	Restriction (j) was not satisfied.	
3610	Restriction (k) was not satisfied.	
3620	Restriction (l) was not satisfied.	
3700	Restriction (m) was not satisfied.	
3710	Restriction (n) was not satisfied.	
3720	Restriction (o) was not satisfied.	
3800	Restriction (p) was not satisfied.	
3810	Restriction (q) was not satisfied.	
3820	Restriction (r) was not satisfied.	
3900	Restriction (s) was not satisfied.	
3910	Restriction (t) was not satisfied.	
3920	Restriction (u) was not satisfied.	
4000	The normalized equation could not be solved.	

(6) **Notes**

(a) Different B–splines may be obtained according to the knot values.

(b) Number of internal knots is $(\mathrm{nxd}-\mathrm{mx}, \mathrm{nyd}-\mathrm{my}, \mathrm{nzd}-\mathrm{mz})$.

(c) This library provides both single–precision and double–precision functions, the double–precision functions should be used for higher precision of solutions.

(7) **Example**

(a) Problem
Perform the fitting to the following data f_{ijk} using a cubic spline function.

$$f_{ijk} = 10 + \frac{1}{0.03+(x_i-0.5)^2} + \frac{1}{0.04+(y_j-0.2)^2} + \frac{1}{0.05+(z_k-0.6)^2} + e_{ijk}$$
$$(x_i = 0.0, 0.1, \cdots, 0.8; y_j = 0.0, 0.1, \cdots, 0.8; z_k = 0.0, 0.1, \cdots, 0.8)$$

Here, e_{ijk} are mutually independent errors that obey a normal distribution having mean 0 and variance 0.01.

(b) Input data

Sample points (xd, yd, zd, fd), nxd=9, nyd=9, nzd=9,

knots in x direction xk, mx=4,

knots in y direction yk, my=4,

knots in z direction zk, mz=4,

x coordinates for calculating interpolation values xx, nnx=9,

y coordinates for calculating interpolation values yy, nny=9,

z coordinates for calculating interpolation values zz and nnz=9.

(c) Main Program

```
/*      C interface example for ASL_dgiss3 */

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

int main()
{
    double *xd;
    int nxd;
    double *yd;
    int nyd;
    double *zd;
    int nzd;
    double *fd;
    double *xk;
    int mx;
    double *yk;
    int my;
    double *zk;
    int mz;
    double *xx;
    int nnx;
    double *yy;
    int nny;
    double *zz;
    int nnz;
    double *s;
    double *rs;
    double aic;
    double *wk;
    double *e,am,sg;
    int *iwk;
    int ierr;
    double ff,sumt2,sumd2,fit;
    int i,j,k,ix,iy;
    FILE *fp;

    fp = fopen( "dgiss3.dat", "r" );
    if( fp == NULL )
    {
        printf( "file open error\n" );
        return -1;
    }

    printf( "    *** ASL_dgiss3 ***\n" );
    printf( "\n     ** Input **\n\n" );

    fscanf( fp, "%d", &nxd );
    fscanf( fp, "%d", &mx );
    fscanf( fp, "%d", &nnx );
    fscanf( fp, "%d", &nyd );
    fscanf( fp, "%d", &my );
    fscanf( fp, "%d", &nny );
    fscanf( fp, "%d", &nzd );
    fscanf( fp, "%d", &mz );
    fscanf( fp, "%d", &nnz );

    iwk = ( int * )malloc((size_t)( sizeof(int) *
        (nxd+nyd+nzd+nxd*nyd*nzd) ));
    if( iwk == NULL )
    {
        printf( "no enough memory for array iwk\n" );
        return -1;
    }

    xd = ( double * )malloc((size_t)( sizeof(double) * nxd ));
```

```
if( xd == NULL )
{
    printf( "no enough memory for array xd\n" );
    return -1;
}

yd = ( double * )malloc((size_t)( sizeof(double) * nyd ));
if( yd == NULL )
{
    printf( "no enough memory for array yd\n" );
    return -1;
}

zd = ( double * )malloc((size_t)( sizeof(double) * nzd ));
if( zd == NULL )
{
    printf( "no enough memory for array zd\n" );
    return -1;
}

fd = ( double * )malloc((size_t)( sizeof(double)
    * nxd * nyd * nzd));
if( fd == NULL )
{
    printf( "no enough memory for array fd\n" );
    return -1;
}

xk = ( double * )malloc((size_t)( sizeof(double) * (nxd+mx) ));
if( xk == NULL )
{
    printf( "no enough memory for array xk\n" );
    return -1;
}

yk = ( double * )malloc((size_t)( sizeof(double) * (nyd+my) ));
if( yk == NULL )
{
    printf( "no enough memory for array yk\n" );
    return -1;
}

zk = ( double * )malloc((size_t)( sizeof(double) * (nzd+mz) ));
if( zk == NULL )
{
    printf( "no enough memory for array zk\n" );
    return -1;
}

xx = ( double * )malloc((size_t)( sizeof(double) * nnx ));
if( xx == NULL )
{
    printf( "no enough memory for array xx\n" );
    return -1;
}

yy = ( double * )malloc((size_t)( sizeof(double) * nny ));
if( yy == NULL )
{
    printf( "no enough memory for array yy\n" );
    return -1;
}

zz = ( double * )malloc((size_t)( sizeof(double) * nnz ));
if( zz == NULL )
{
    printf( "no enough memory for array zz\n" );
    return -1;
}

s = ( double * )malloc((size_t)( sizeof(double) * nnx * nny * nnz));
if( s == NULL )
{
    printf( "no enough memory for array s\n" );
    return -1;
}

rs = ( double * )malloc((size_t)( sizeof(double)
    * nxd * nyd * nzd ));
if( rs == NULL )
{
    printf( "no enough memory for array rs\n" );
    return -1;
}

e = ( double * )malloc((size_t)( sizeof(double)
    * nxd * nyd * nzd ));
if( e == NULL )
{
    printf( "no enough memory for array e\n" );
    return -1;
}
```

```
wk = ( double * )malloc((size_t)( sizeof(double)*
(nxd*nxd*nyd*nyd*nzd*nzd+nxd*nyd*nzd
 +nxd*nxd*nyd*nyd+nxd*nxd+mx*(2+nxd)
 +my*(2+nyd)+2*mz+3) ));
if( wk == NULL )
{
    printf( "no enough memory for array wk\n" );
    return -1;
}

printf( "\tnxd = %6d mx = %6d nnx = %6d\n",
    nxd, mx, nnx );
printf( "\tnyd = %6d my = %6d nny = %6d\n",
    nyd, my, nny );
printf( "\tnzd = %6d mz = %6d nnz = %6d\n",
    nzd, mz, nnz );

for( i=0 ; i<nxd ; i++ )
{
    xd[i] = 0.1 * (double)i;
}
for( j=0 ; j<nyd ; j++ )
{
    yd[j] = 0.1 * (double)j;
}
for( k=0 ; k<nzd ; k++ )
{
    zd[k] = 0.1 * (double)k;
}

ix=1;
iy=1;
am = 0.0;
sg = 0.1;

ierr = ASL_djdbno(nxd*nyd*nzd, am, sg, &ix, &iy, e);

for( k=0 ; k<nzd ; k++ )
{
for( j=0 ; j<nyd ; j++ )
{
for( i=0 ; i<nxd ; i++ )
{
    fd[i+j*nxd+k*nxd*nyd] = 10.0
    +1.0/(0.03+(xd[i]-0.5)*(xd[i]-0.5))
    +1.0/(0.04+(yd[j]-0.2)*(yd[j]-0.2))
    +1.0/(0.05+(zd[k]-0.6)*(zd[k]-0.6))
    +e[k*nxd*nyd+j*nxd+i];
}
}
}

for( i=0 ; i<nxd+mx ; i++ )
{
    fscanf( fp, "%lf", &xk[i] );
}
for( j=0 ; j<nyd+my ; j++ )
{
    fscanf( fp, "%lf", &yk[j] );
}
for( k=0 ; k<nzd+mz ; k++ )
{
    fscanf( fp, "%lf", &zk[k] );
}

for( i=0 ; i<nnx ; i++ )
{
    xx[i] = 0.1 * (double)i;
}
for( j=0 ; j<nny ; j++ )
{
    yy[j] = 0.1 * (double)j;
}
for( k=0 ; k<nnz ; k++ )
{
    zz[k] = 0.1 * (double)k;
}

fclose( fp );

ierr = ASL_dgiss3(xd, nxd, yd, nyd, zd, nzd, fd, xk, mx, yk, my,
     zk, mz, xx, nnx, yy, nny, zz, nnz, s, rs, &aic, wk, iwk);

printf( "\n     ** Output **\n" );
printf( "\n\tierr = %6d\n", ierr );

if( ierr < 3000 )
{
    printf( "\n\taic  = %8.3g\n", aic );
    sumt2 = 0.0;
```

```
            sumd2 = 0.0;
            for( i=0 ; i<nnx ; i++ )
            {
            for( j=0 ; j<nny ; j++ )
            {
            for( k=0 ; k<nnz ; k++ )
            {
                ff = 10.0
                   +1.0/(0.03+(xd[i]-0.5)*(xd[i]-0.5))
                   +1.0/(0.04+(yd[j]-0.2)*(yd[j]-0.2))
                   +1.0/(0.05+(zd[k]-0.6)*(zd[k]-0.6));
                sumt2 += ff * ff;
                sumd2 += (ff-s[i+j*nnx+k*nnx*nny])
                       * (ff-s[i+j*nnx+k*nnx*nny]);
            }
            }
            }
            sumt2 = sqrt(sumt2);
            sumd2 = sqrt(sumd2);
            fit = 100.0;
            if( ( sumt2 != 0.0 )||( sumd2 != 0.0 ) )
            {
                fit = ( sumt2 / ( sumt2 + sumd2 ) ) * 100.0;
            }
            printf( "\n\tFitting rate = %8.3g\n", fit );
        }

        free( iwk );
        free( xd );
        free( yd );
        free( zd );
        free( fd );
        free( xk );
        free( yk );
        free( zk );
        free( xx );
        free( yy );
        free( zz );
        free( s );
        free( rs );
        free( e );
        free( wk );

        return 0;
    }
```

(d) Output results

```
        *** ASL_dgiss3 ***

         ** Input **

        nxd =      9 mx =      4 nnx =      9
        nyd =      9 my =      4 nny =      9
        nzd =      9 mz =      4 nnz =      9

         ** Output **

        ierr =      0

        aic  = -3.98e+03

        Fitting rate =     99.8
```

Appendix A

MACHINE CONSTANTS USED IN ASL C INTERFACE

A.1 Units for Determining Error

The table below shows values in ASL C interface as units for determining error in floating point calculations. The units shown in the table are numeric values determined by the internal representation of floating point data. ASL C interface uses these units for determining convergence and zeros.

Table A−1 Units for Determining Error

Single-precision	Double-precision
$2^{-23}(\simeq 1.19 \times 10^{-7})$	$2^{-52}(\simeq 2.22 \times 10^{-16})$

Remark: The unit for determining error ε, which is also called the machine ε, is usually defined as the smallest positive constant for which the calculation result of $1+\varepsilon$ differs from 1 in the corresponding floating point mode. Therefore, seeing the unit for determining error enables you to know the maximum number of significant digits of an operation (on the mantissa) in that floating point mode.

A.2 Maximum and Minimum Values of Floating Point Data

The table below shows maximum and minimum values of floating point data defined within ASL C interface. Note that the maximum and minimum values shown below may differ from the maximum and minimum values that are actually used by the hardware for each floating point mode.

Table A−2 Maximum and Minimum Values of Floating Point Data

	Single-precision	Double-precision
Maximum value	$2^{127}(2-2^{-23})$ $(\simeq 3.40 \times 10^{38})$	$2^{1023}(2-2^{-52})$ $(\simeq 1.80 \times 10^{308})$
Positive minimum value	2^{-126} $(\simeq 1.17 \times 10^{-38})$	2^{-1022} $(\simeq 2.23 \times 10^{-308})$
Negative maximum value	-2^{-126} $(\simeq -1.17 \times 10^{-38})$	-2^{-1022} $(\simeq -2.23 \times 10^{-308})$
Minimum value	$-2^{127}(2-2^{-23})$ $(\simeq -3.40 \times 10^{38})$	$-2^{1023}(2-2^{-52})$ $(\simeq -1.80 \times 10^{308})$

Index

(*) SMP Functions = Shared Memory Parallel Processing Functions