April 1990
Order Number: 311868-001

iPSC®/2 and iPSC®/860
MATH LIBRARIES REFERENCE MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	04/90

RESTRICTED RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

PREFACE

This manual describes the C and Fortran iPSC®/2 vector library (VecLib) routines.

This manual uses the term "iPSC system(s)" to refer to the following iSC products: iPSC®/2, iPSC®/2S, iPSC®/860, iPSC®/860S, and iPSC®/860Plus.

This manual assumes that you are an application programmer proficient in the C and Fortran languages and the UNIX operating system.

ORGANIZATION

Chapter 1	Describes the routines that provide a set of basic vector operations that can be called from C routines.
Chapter 2	Describes the routines that provide a set of basic vector operations that can be called from Fortran routines.
Appendix A	Summarizes the C VecLib routines according to function.
Appendix B	Summarizes the Fortran VecLib routines according to function.

APPLICABLE DOCUMENTS

For more information, refer to the following manuals:

iPSC® System Manuals

iPSC®/2 Ada Program Development Guide
Describes the tools available to develop Ada Development programs for the iPSC/2, and tells how to use them.

iPSC®/2 Ada Programmer's Reference Manual
Provides detailed information on all Ada routines and commands for the iPSC/2 system..

iPSC®/2 and iPSC®/860 C Language Reference Manual
Describes the C compiler for the iPSC system.

iPSC®/2 and iPSC®/860 FORGE User's Guide
Tells how to use the FORGE tool set to analyze Fortran programs to port them to a parallel machine.

iPSC®/2 and iPSC®/860 Fortran Language Reference Manual
Describes the Fortran compiler for the iPSC system.

iPSC®/2 and iPSC®/860 Programmer's Reference Manual
Describes iPSC system commands and system calls (for both C and Fortran).

iPSC®/2 and iPSC®/860 System Administrator's Guide
Provides a detailed description of the system administration tasks related to operating and maintaining an iPSC system.

iPSC®/2 and iPSC®/860 User's Guide
Provides an overview of the iPSC system including both hardware and software architectures. Describes how to use the system to develop and run programs.

iPSC®/2 Lisp Language Reference Manual
Describes the Lisp implementation that runs on the iPSC/2 nodes, and its extensions.

iPSC®/2 Lisp Programmer's Reference Manual
Describes the iPSC/2 Lisp user interface and the iPSC/2 Lisp concurrent constructs.

iPSC®/2 DECON User's Guide
Tells how to use DECON, the iPSC/2 concurrent debugger.

iPSC®/2 DECON User's Guide RX Beta Change Notice
Describes RX Beta Changes to DECON commands.

iPSC®/2 Simulator Manual
Tells how to use the iPSC/2 Simulator for software development.

iPSC®/2 VAST2 User's Guide
Provides detailed information on the use of the iPSC/2-VX version of VAST2 software.

iPSC®/2 VX User's Guide
Provides detailed information on the development of programs for the iPSC/2-VX vector processing system.

iPSC®/860 Vector User's Guide
Provides detailed information on the use of iPSC/860 vector processing features and the iPSC/860 version of VAST2 software.

Intel® Manuals

Intel® NFS for System V/386 Programmer's Guide and Reference
Describes the NFS programming environment and provides information on programming tools.

Intel® NFS for System V/386 User's/System Administrator's Guide and Reference
Describes the NFS programming environment, provides user information, and provides system administration information.

Intel® TCP/IP for SYSTEM V/386 Administrator's Guide and Reference
(Replaces Excelan TCP/IP documentation)
Describes TCP/IP Network administration.

Intel® TCP/IP for SYSTEM V/386 Programmer's Guide and Reference Manual
(Replaces Excelan TCP/IP documentation)
Describes the TCP/IP Network programming environment and provides information on programming tools.

Intel® TCP/IP for SYSTEM V/386 User's Guide and Reference
(Replaces Excelan TCP/IP documentation)
Describes the TCP/IP Network programming environment and provides user information.

i860™ 64-Bit Microprocessor Programmer's Reference Manual
Provides detailed information that enables programmers and designers to use the i860 microprocessor.

i860™ Microprocessor Assembler and Linker Reference Manual
Provides detailed information that enables programmers and designers to use the i860 assembler and linker.

SYP301 Installation and User's Guide
Describes installation and startup for the System Resource Manager. Also provides hardware technical data.

Other Manuals

C: A Reference Manual - Harbison and Steele
Describes the C programming language.

Reference Manual For The Ada Programming Language - ANSI/MIL-STD-1815A-1983
Describes the Ada programming language.

The C Programming Language - Kernighan and Ritchie
Describes the C programming language.

UNIX System V Manual Set
Provides a complete description of UNIX System V.

NOTATIONAL CONVENTIONS

This section describes the following notational conventions:

- Type style usage
- Examples in text
- Command syntax
- System call syntax

Type Style Usage

This manual uses the following type style conventions:

- **Bold** type style is used for anything that must be entered exactly as shown, including:

  Command names (including absolute pathname versions)
  Switches, flags, and options
  System Call names
  Routine names (predefined *and* user-defined)
  Reserved words

- *Italic* type style is used for:

  Variables
  File names (including absolute pathname versions)
  Directory names
  Process names
  User names
  Emphasis

- ***Bold-italic*** type style is used for user input described in text (what you enter in response to some prompt).

- `Plain-Monospace` type style is used for:

  Computer output (prompts and messages)
  Code
  Error messages
  Values of variables

- ***Bold-Italic-Monospace*** type style is used for user input displayed in an example (what you enter in response to some prompt)

- **Bold-Monospace** type style is used for the names of keyboard keys (which are also enclosed in angle brackets). For example:

<Alt>	**<Backspace>**
<Break>	**<Ctrl>**
<Delete> or **<Del>**	**<Enter>**
<Esc>	**<Tab>**

  For a key that prints, the result of pressing the key is used. For example, **s** means press the key labeled **<S>**, and **S** means press and hold the **<Shift>** key while pressing the **<S>** key.

  A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. This has the effect of producing a single keystroke (which is why there is only one set of angle brackets). For example:

```
<Ctrl-S>
<Ctrl-@>
<Ctrl-Alt-Del>
```

  Note that there are several ways to reference some keys. For example, **S** and **<Shift-s>** mean the same thing. In such cases, the simplest method (**S**) is usually used.

Examples in Text

This manual uses `monospace` type style for examples.

In examples of interactive sessions, user input appears in ***bold-italic-monospace*** type style to distiguish it from computer output. For example:

```
# getcube -t20                                          (userinput)
getcube successful: cube type 2m8n0 allocated          (computer output)
```

Many examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style.

Command Syntax

You enter commands at the console. Commands have the following form:

```
command arguments
```

where:

command	Is a keyword (see following section).
arguments	Consist of zero or more optional or required terms. One kind of argument, called a switch, consists of a dash followed by one or more letters. Sometimes a name or value follows the switch.

Command syntax is represented in this manual as follows:

keywords	Keywords appear in bold type. You must use keywords exactly as written. Command names and switches are keywords.
variables	Variables appear in italic type. They represent arguments that you must supply (in the place of the italicized word) when you invoke the command. Do not enter the name of the italicized word itself.
`[ ]`	Optional command-line arguments appear within square brackets. If the optional argument is a keyword, it appears in bold type. If the argument represents a variable, it appears in italic type.
`...`	Ellipses (three periods in a row) following an argument indicate that you may repeat that argument.
`|`	A vertical bar separates two or more choices of which you may select only one. For example, `[-c cubename | -a ]` indicates that may select the first option (`-c cubename`) or the second option (`-a`).

The following command syntax for the **load** command shows all of the syntax elements:

```
load [ -c cubename | -a ] [ -p pid ] [ -H ] [ node... ]
   filename [ arguments... ]
```

According to the syntax, only the word **load** and a *filename* are required. However, five options are available: four options require a switch, and two options (*node* and *arguments*), allow more than one argument of this kind. The following example of this command uses the *cubename* option, two *node* values, an input file name "input_file," and an *argument* value of 1000 (this example does not use the **-p** and **-H** options):

```
load -c alpha 3 4 input_file 1000
```

System Call Syntax

You can use iPSC/2 system calls in programs in the same way that you use other system calls. In this manual, system calls are described as follows:

Synopsis

return_type **name** (*parameters*)

Parameter Declarations

type *parameter*

Note that the return type appears in regular type, the name of the call appears in bold type, the data type of the parameters appears in bold type, and the parameters appear in italic type. For example, the following is the synopsis and parameter declarations for **getcube()** for C programs:

```
getcube(cubename, cubetype, srmname, keep)

char   *cubename;
char   *cubetype;
char   *srmname;
long   keep;
```

The Fortran syntax for the same command appears as follows:

```
SUBROUTINE GETCUBE(cubename, cubetype, srmname, keep)

CHARACTER  cubename*(*)
CHARACTER  cubetype*(*)
CHARACTER  srmname*(*)
INTEGER    keep
```

C system calls that have no return value do not have a word preceding the name of the call. Fortran routines that have no return value have the word SUBROUTINE in place of the return type. Fortran routines that return a value have the word "FUNCTION" following the return type. For example, **iprobe()**, which returns an integer, appears as follows for the two languages:

In Fortran:
```
INTEGER FUNCTION IPROBE(id)
```

In C:
```
long iprobe(id)
```

TABLE OF CONTENTS

CHAPTER 1
C VECLIB ROUTINES

CHAPTER 2
FORTRAN VECLIB ROUTINES

LIST OF TABLES

C VECLIB ROUTINES 1

INTRODUCTION

The Vector Library (VecLib) contains routines that support Intel's numeric products. These routines provide a basic set of vector operations that can be called from C and Fortran routines. From within C programs, you can call these routines to replace C *for* loops. There are six versions of the VecLib routines contained in the following libraries:

Library	Implemented With	Function
/usr/lib/libvec.a	Fortran and C	Executes on standard node board or system resource manager.
/usr/lib/libsxvec.a	Fortran and C	Executes on SX option board.
/usr/lib/libvxvec.a	Microcode	Executes on vector processor board.
/usr/lib/libvxdbvec.a	Microcode	Executes on vector processor board. Provides debug checks.
/usr/lib/libvxsxvec.a	Microcode	Executes on vector processor board and nodes that have SX option.
/usr/lib/libvxsxdbvec.a	Microcode	Executes on vector processor board and nodes that have SX option. Provides debug checks.

The */usr/lib/libsxvec.a* library can be used only if the SX option is installed in your system. The */usr/lib/libvxsxvec.a* and */usr/lib/libvxsxdbvec.a* libraries can be used only if both the SX option and vector processor boards are installed. The last four libraries (those with *vx* in the names) can be used only if vector processor boards are installed. (Refer to the *iPSC®/2 VX User's Guide* for more information.)

This chapter contains the following information:

- The conventions used in naming routines.
- Conventions used for parameter names in the reference pages (formal parameters).
- General rules and information that will help you use the VecLib routines.
- C macros for mathematical operations used in the routines.
- General limitations.
- Error messages.
- A table that is a summary of all of the routines, listing a brief description of each routine along with a synopsis of the call and the C equivalent of the call.
- The final section contains a reference page for each VecLib routine, in order alphabetically by root name. For each routine, the reference contains a brief definition, the calling sequence, the C equivalent, and a mathematical description.

NAMING CONVENTIONS

Routines exist for both single precision and double precision functions. In some cases, routines exist for integer, logical, and complex functions. The data type a function operates on is indicated by the first or second letter of the routine name; "d" represents double, "s" represents single precision (*float*), and "i" represents integer. In addition, three special types are defined: "l" represents logical, "c" represents complex, and "z" represents double precision complex (*zcomplex*). For example:

Root	Double	Float	Logical	Integer	Complex	Zcomplex
*x*copy	dcopy	scopy	lcopy	icopy	ccopy	zcopy

The vector library includes most of the Basic Linear Algebra Subprograms (BLAS) and uses the BLAS calling sequence and naming conventions.

NOTE

For more information on the Basic Linear Algebra Subprograms, refer to the following publications:

Bunch, J.R., Dongarra, J.J., Moler, C.B., Stewart, G.W., *LINPACK User's Guide*, 1979, Appendix A.

Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T., "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Software*, 1979, Vol. 5, No. 3, pp. 308-371.

Some of the vector routines use the Reverse Polish Notation (RPN) naming convention. An example is shown in Table 1-1.

Table 1-1. Reverse Polish Notation

Routine	Notation	Description
*x*svtvm	s = scalar v = vector t = times m = minus	Scalar times vector quantity minus vector
*x*svvtp	s = scalar v = vector t = times p = plus	Scalar plus the quantity of vector times vector

FORMAL PARAMETERS

The calling sequence for each routine always includes the number of iterations as the first parameter, followed by any scalar and vector values. Vector arguments in the parameter list are in **bold** type style to distinguish them from scalar arguments. Each vector argument is followed immediately in the argument list by its increment or "stride" (i.e., z, incz). Notice that some routines have an output vector while other routines overwrite one of the input vectors.

Routines that take one vector as input use "**x**." Routines that use two vectors use "**x**" and "**y**." Routines that use three vectors use "**x**", "**y**", and "**z**." Routines that use four vectors use "**w**", "**x**", "**y**", and "**z**."

Routines that use one scalar use "alpha". Routines that use two scalars use "alpha" and "beta".

In general, formal parameters can be described as follows:

n	number of vector elements to process
x	vector with at least (n-1) * ABS(incx)+1 elements
incx	stride of the vector **x**
alpha	scalar constant
beta	scalar constant

GENERAL RULES

- All floating point arguments and results normally match the *type* (for example, double or float) of the subprogram.
- Each routine, as documented in this chapter, includes three parameters:
  - the number of iterations
  - vector and scalar values, if appropriate
  - strides (increments)
- In the calling sequence, if the vector type differs from the type indicated by the routine name, it is indicated by a single letter preceded by an "s" for single, "d" for double, "i" for integer, "l" for logical, "c" for complex, or "z" for double precision complex type vectors. For example, the vector "lz" in the call below is of type *logical*; "x" and "y" are of type *float*.

```
sgt (n, x, incx, y, incy, lz, incz);
```

- The logical (*logical*), complex (*complex*) and double precision complex (*zcomplex*) types are defined with *typedefs* in the include file *cveclib.h*. These types behave as the Fortran logical, complex, and double precision complex types do. You can declare variables to be of these types, declare arrays of them, and use them in structures. The *logical* type is defined such that FALSE is equivalent to 0, and TRUE is equivalent to 1.

  In the reference pages, the C equivalents for routines that use *complex* and *zcomplex* types assume that the simple mathematical operations (+, -, *, /, unary minus, ==, !=) are supported; in fact, if you were to use the equivalent statements, you would substitute the functions shown in the following section for these operations. The following section also lists definitions for the ABS, MIN, and MAX functions.
- The trigonometric routines (*x*VATN2, *x*COS, *x*SIN) expect their arguments to be in radians, or return a vector whose values are expressed in radians.
- Limitations specified in some routines are not checked at runtime. If violated, erroneous results are produced. The only routines which provide some parameter-checking are in *libvxdbvec.a* or *libvxsxdbvec.a*.
- Unless otherwise noted, vector elements that are not contiguous but are separated by equal distances (either positive or negative) can be accessed by using a stride other than one. Non-unit strides follow the BLAS parameter-passing convention, which does not allow negative indices to arrays.

The formula for indexing through the vectors [*n,x[lowest_index],incx*] for *i* from 0 to *n*-1 is:

```
if (incx > 0) x[i * incx + lowest_index];
else    x[(i - n + 1) * incx + lowest_index];
```

The following example defines two arrays, *x* and *y*. Then, there are three calls to **saxpy** (a routine that multiplies a scalar times a vector, adds the result to another vector, and then overwrites the second vector with the result). These calls specify positive and negative strides not equal to 1. For each, the results in *y* are shown.

```
float x[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
float y[10]={.01,.02,.03,.04,.05,.06,.07,.08,.09,.10}; float
a = 1.0

saxpy (3,&a,x,2,y,1);
```

y will be equal to {1.01, 3.02, 5.03, .04, .05, .06, .07, .08, .09, .10}

```
saxpy (3,&a,x,-2,y,1);
```

y will be equal to {5.01, 3.02, 1.03,.04, .05, .06, .07, .08, .09, .10}

```
saxpy (3,&a,x[2],-2,y[1],2);
```

y will be equal to {.01, 7.02, .03, 5.04, .05, 3.06, .07, .08, .09, .10}

DEFINITIONS AND FUNCTIONS FOR COMPLEX OPERATIONS

The complex and double precision complex types (**complex** and **zcomplex**) are defined by **typedefs** in the include file *cveclib.h*. The definitions are as follows:

```
typedef enum{FALSE,TRUE;} logical;
typedef struct {float r,i;} complex;
typedef struct {double r,i} zcomplex;
```

In the same file, the MIN, MAX, ABS, and CABS operations (used in the C equivalent descriptions in the reference pages) are defined as follows:

```
#define MIN(x,y)  (x)<(y)(x):(y)
#define MAX(x,y)  (x)>(y)(x):(y)
#define ABS(x)  (x)<0-(x):(x)
#define CABS(x)  sqrt(x.r*x.r+x.i*x.i)
```

MIN, MAX, and ABS are used for integer, float, and double routines. The C equivalents of the routines on the reference pages assume that the complex types support the simple operations +, -, *, /,unary minus, =, ==, and !=. In fact, if you were to use the equivalents in a program, you would need to substitute the following functions for these operations (functions are shown for the complex type; for double precision complex, substitute **zcomplex** for complex and **double** for float).

Addition:

```
complex add(a,b)
complex a,b;
{
complex c;
c.r = a.r + b.r;
c.i = a.i + b.i;
return c;
}
```

Subtraction:

```
complex sub(a,b)
complex a,b;
{
complex c;
c.r = a.r - b.r;
c.i = a.i - b.i;
return c;
}
```

Multiplication:

```
complex mul(a,b)
complex a,b;
{
complex c;
c.r = a.r*b.r - a.i*b.i;
c.i = a.r*b.r + a.i*b.i;
return c;
}
```

Conjugation:

```
complex CONJG(a)
complex a;
{
complex b;
b.r = a.r;
b.i = -a.i;
return b;
}
```

Division:

```
complex div(a,b)
complex a,b;
{
   complex c;
   c.r = (a.r * b.r + a.i * b.i)/(b.r * b.r + b.i * b.i);
   c.i = (a.r * b.r - a.i * b.i)/(b.r * b.r + b.i * b.i);
   return c;
}
```

Negative:

```
complex neg(a)
complex a;
{
complex b;
b.r = -a.r;
b.i = -a.i;
return b;
}
```

Complex from two reals (CMPLX):

```
complex CMPLX(a,b)
float a,b;
{
complex c;
c.r = a;
c.i = b;
return c;
}
```

Equal:

```
int eq(a,b)
complex a;
{
return(a.r==b.r)&&(a.i==b.i);
}
```

Not Equal:

```
int ne(a,b)
complex a;
{
return(a.r!=b.r)(a.i!=b.i);
}
```

GENERAL LIMITATIONS

The following limitations apply to vector routines described in this chapter.

- C defines logical true to be nonzero and logical false to be zero. The VecLib routines use `1` for true and `0` for false. Before you use a vector in a VecLib routine that expects a logical vector as input, you should convert the vector to make sure that all "true" values are `1`, and not some other nonzero integer. You can do this easily and quickly with the **isne()** routine (see *x***sne**).

- For *n* less than or equal to zero, voids do nothing, and most functions return `0`. However, the indexing functions return `-1` to indicate a meaningless value (**ixamin()**, **ixamax()**, **ixmin()**, **ixmax()**, **ifirst()**, **ilast()**).

- Output vector strides of zero produce undefined results if *n* is greater than 1.

- Care must be used when overlapping input and output vectors. Because of the pipelined nature of the VX arithmetic, the results for one iteration (*i*) are not always written to memory before the next iteration (*i*+1) is begun.

- Fortran and C versions of libraries may not produce exactly the same results as microcode versions.

  - 387™ coprocessor supports denormals, VX and SX hardware does not.

  - VX hardware does not support the IEEE divide or square root operations.

  - 387, VX, and SX coprocessors use different algorithms for transcendental functions.

  - Because VX arithmetic hardware is pipelined, accumulation of round-off error for reduction functions is not the same as for 387 or SX coprocessors.

ERROR MESSAGES

Error messages are generated only when running VecLib routines in *libvxdbvec.a* or *libvxsxdbvec.a*. In these messages, "<x>" represents vector or scalar parameter names and "<ddot>"represents any of the VecLib routine names.

`VX veclib: Error in length for argument <x> in call to <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.

Action: Call iSC Customer Support

`VX veclib: Error in memory space for argument <x> in call to <ddot>`

Description: The data is not on vector board memory but is on the node board.

Cause: Incorrect use of *vxld*.

Action: Move the data containing the vector to the VX memory using *vxld*.

`VX veclib: Error in physical alignment for argument <x> in call to <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.

Action: Call iSC Customer Support

`VX veclib: Error in physical bounds check for argument <x> in call to <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.

Action: Call iSC Customer Support

`VX veclib: Error in type for check of argument <x> in call from <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.

Action: Call iSC Customer Support

`VX veclib: Error in virtual alignment for argument <x> in call to <ddot>`

Description: The virtual address modulo the data size (in bytes) is not equal to zero.

Cause: Ignored warning from compiler about equivalence misalignment or compiler or linker didn't align the data correctly.

Action: If warning ignored, change equivalence statement. If compiler or linker didn't align data correctly, call iSC Customer Support.

`VX veclib: Error in virtual bounds check for argument <x> in call to <ddot>`

Description: The data is not contiguous in virtual memory.

Cause: Requesting vector operations outside the bounds of an array.

Action: Locate error in your Fortran program.

```
VX veclib: The following enabled exceptions have occurred: NaN (not-a-number),
           overflow, underflow.
```

Cause: Requested modification of exceptions using *vpexcept* and algorithm-produced exception.

Action: Modify algorithm used so exception does not occur.

*x*ASUM() *x*ASUM()

Sum of the absolute values. (Function returning a value.)

Calling Sequence

Double precision:

```
double d, x[], dasum();
int n, incx;

d = dasum(n, x, incx);
```

Single precision:

```
float s, x[], sasum();
int n, incx;

s = sasum(n, x, incx);
```

C Equivalent

```
dasum = 0.0;
for(i = 0; i < n; i++)
   dasum += ABS(x[i]);
```

Description

$$\text{dasum} = \sum_{i=0}^{n-1} |x[i]|$$

*x*AXPY() *x*AXPY()

Scalar times a vector plus a vector to itself.

Calling Sequence

Double precision:

```
double x[], alpha, y[];
int n, incx, incy;

daxpy(n, &alpha, x, incx, y, incy);
```

Single precision:

```
float x[], alpha, y[];
int n, incx, incy;

saxpy(n, &alpha, x, incx, y, incy);
```

Complex:

```
complex x[], alpha, y[];
int n, incx, incy

caxpy(n, &alpha, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], alpha, y[];
int n, incx, incy;

zaxpy(n, &alpha, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < n; i++)
  y[i] += alpha * x[i];
```

*x*CLIP() *x*CLIP()

Clip to interval [alpha, beta].

Calling Sequence

Double precision:

```
double x[], alpha, y[], beta;
int n, incx, incy;

dclip(n, x, incx, &alpha, &beta, y, incy);
```

Single precision:

```
float x[], alpha, y[], beta;
int n, incx, incy;

sclip(n, x, incx, &alpha, &beta, y, incy)
```

C Equivalent

```
for(i = 0; i < n; i++)
  y[i] = MIN(MAX(x[i], alpha), beta);
```

Description

$$y[i] = \begin{cases} \texttt{alpha} & \text{if } \texttt{x[i]} < \texttt{alpha} \\ \texttt{x[i]} & \text{if } \texttt{alpha} \le \texttt{x[i]} \le \texttt{beta} \\ \texttt{beta} & \text{if } \texttt{x[i]} > \texttt{beta} \end{cases}$$

Note that "alpha" must be less than or equal to "beta".

*x*CNDST() *x*CNDST()

Conditional assignment.

Calling Sequence

Double precision:

```
double x[], z[];
logical ly[];
int n, incx, incy, incz;

dcndst(n, x, incx, ly, incy, z, incz);
```

Single precision:

```
float x[], z[];
logical ly[];
int n, incx, incy, incz;

scndst(n, x, incx, ly, incy, z, incz);
```

C Equivalent

```
for (i = 0; i < n; i++)
      if (ly[i]) z[i] = x[i];
```

*x*COPY() *x*COPY()

Copy vector.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dcopy(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

scopy(n, x, incx, y, incy);
```

Integer:

```
int x[], y[];
int n, incx, incy;

icopy(n, x, incx, y, incy);
```

Logical:

```
logical x[], y[];
int n, incx, incy;

lcopy(n, x, incx, y, incy);
```

Complex:

```
complex x[], y[];
int n, incx, incy;

ccopy(n, x, incx, y, incy);
```

*x*COPY() *(cont.)*

*x*COPY() *(cont.)*

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;
```

C Equivalent

```
zcopy(n, x, incx, y, incy);
for (i = 0; i < n; i++)
      y[i] = x[i];
```

*x*DOT() *x*DOT()

Dot product of two vectors. (Function returning a value.)

Calling Sequence

Double precision:

```
double d, x[], y[], ddot();
int n, incx, incy;

d = ddot(n, x, incx, y, incy);
```

Single precision:

```
real s, x[], y[], sdot();
int n, incx, incy;

s = sdot(n, x, incx, y, incy);
```

C Equivalent

```
ddot = 0.0;
for (i = 0; i < n; i++);
  ddot = ddot + x[i] * y[i];
```

Description

$$\underline{d}dot = \sum_{i=0}^{n-1} (x[i] \bullet y[i])$$

*x*DOTC() *x*DOTC()

Dot product of two complex vectors, with the first vector being conjugated. (Function returning a value.)

Calling Sequence

Complex:

```
complex c, x[], y[], cdotc();
int n, incx, incy;

c = cdotc(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex z, x[], y[], zdotc();
int n, incx, incy;

z = zdotc(n, x, incx, y incy);
```

C Equivalent

```
cdotc = CMPLX(0.0, 0.0);
for(i = 0; i < n; i++)
      cdotc = cdotc + (CONJG(x[i]) * y[i]);
```

Description

$$\underline{c}dotc = \sum_{i=0}^{n-1} (CONJG(x[i]) \cdot y[i])$$

*x*DOTU() *x*DOTU()

Dot product of two complex vectors. (Function returning a value.)

Calling Sequence

Complex:

```
complex c, x[], y[], cdotu();
int n, incx, incy;

c = cdotu(n, x, incx, y, incy)
```

Double precision complex:

```
zcomplex z, x[], y[], zdotu();
int n, incx, incy;

z = zdotu(n, x, incx, y, incy);
```

C Equivalent

```
cdotu = CMPLX(0.0, 0.0);
for (i = 0; i < n; i++)
      cdotu = cdotu + x[i] * y[i];
```

Description

$$\underline{c}\text{dotu} = \sum_{i=0}^{n-1} x[i] \cdot y[i]$$

DZASUM() DZASUM()

Sum of the absolute values of the real and imaginary parts of a complex vector. (Function returning a value.)

Calling Sequence

Double precision:

```
double d, dzasum();
zcomplex x[];
int n, incx

d = dzasum(n,. x, incx)
```

C Equivalent

```
dzasum = 0.0;
for(i = 0; i < n; i++)
      dzasum = dzasum + ABS(x[i].r) + ABS(x[i].i);
```

Description

$$\text{dzasum} = \sum_{i=0}^{n-1} |x[i].r| + |x[i].i|$$

*x*EQ() *x*EQ()

Vector element equality.

Calling Sequence

Double precision:

```
double x[], y[];
logical lz[];
int n, incx, incy, incz;

deq(n, x, incx, y, incy, lz, incz);
```

Single precision:

```
float x[], y[];
logical lz[];
int n, incx, incy, incz;

seq(n, x, incx, y, incy, lz, incz);
```

Integer:

```
int x[], y[];
logical lz[];
int n, incx, incy, incz;

ieq(n, x, incx, y, incy, lz, incz);
```

C Equivalent

```
for(i = 0; i < n; i++)
      lz[i] = x[i] == y[i];
```

*x*FFT() *x*FFT()

Fast Fourier Transform.

Calling Sequence

Complex:

```
complex x[], y[];
int n, incx, incy;

cfft(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;

zfft(n, x, incx, y, incy);
```

C Equivalent

```
y = fft(x)
```

Description

This routine performs a one-dimensional **fft** with a maximum vector length of 16384; *n* must be a power of two. The vectors **x** and **y** may not overlap because the transform is not done in place.

The **cfft** routine uses **vpfast_** as a scratch vector.

The **zfft** routine dynamically allocates a buffer of (3*n*+1) eight-byte words the first time you call it. This buffer is reused if subsequent calls to **zfft** use a value of *n* less than or equal to the first, making this routine very fast. If a value of *n* that is larger than the currently allocated buffer is used, the buffer is freed and a larger one is allocated.

If either *incx* or *incy* is not 1, **zfft** will run slower. If both *incx* or *incy* are not 1, **zfft** allocates an additional 2*n* eight-byte words.

*x*FILL() *x*FILL()

Fill vector with scalar.

Calling Sequence

Double precision:

```
double alpha, x[];
int n, incx;

dfill(n, &alpha, x, incx);
```

Single precision:

```
float alpha, x[];
int n, incx;

sfill(n, &alpha, x, incx);
```

Integer:

```
int alpha, x[];
int n, incx;

ifill(n, &alpha, x, incx);
```

Logical:

```
logical alpha, x[];
int n, incx;

lfill(n, &alpha, x, incx);
```

Complex:

```
complex alpha, x[];
int n, incx;

cfill(n, &alpha, x, incx);
```

*x*FILL() *(cont.)*

*x*FILL() *(cont.)*

Double precision complex:

```
zcomplex alpha, x[];
int n, incx;

zfill(n, &alpha, x, incx);
```

C Equivalent

```
for(i = 0; i < n; i++)
      x[i] = alpha;
```

*x*FOLR() *x*FOLR()

First-order linear recurrence routine.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dfolr(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

sfolr(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < n; i++)
      z[i + 1] = y[i] + z[i] * x[i];
```

Note that the routine accesses one more element in *z* than in *w* or *x*. Be sure to make the *z* array large enough (including the increment size) to avoid overwriting other memory.

*x*GATHR() *x*GATHR()

Vector gather.

Calling Sequence

Double precision:

```
double x[], z[];
int n, incx, incy, incz, iy[];

dgathr(n, x, incx, iy, incy, z, incz);
```

Single precision:

```
float x[], z[];
int n, incx, incy, incz, iy[];

sgathr(n, x, incx, iy, incy, z, incz);
```

Integer:

```
int x[], z[];
int n, incx, incy, incz, iy[];

igathr(n, x, incx, iy, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < n; i++)
      z[i] = x[iy[i]];
```

Description

The array *iy* must contain values between 0 and *n*-1.

*x*GE() *x*GE()

Vector element greater than or equal to vector element.

Calling Sequence

Double precision:

```
double x[], y[];
logical lz[];
int n, incx, incy, incz;

dge(n, x, incx, y, incy, lz, incz);
```

Single precision:

```
float x[], y[];
logical lz[];
int n, incx, incy, incz;

sge(n, x, incx, y, incy, lz, incz);
```

Integer:

```
int x[], y[];
logical lz[];
int n, incx, incy, incz;

ige(n, x, incx, y, incy, lz, incz);
```

C Equivalent

```
for(i = 0; i < n; i++)
      lz[i] = x[i] >= y[i];
```

*x*GT() *x*GT()

Vector element greater than vector element.

Calling Sequence

Double precision:

```
double x[], y[];
logical lz[];
int n, incx, incy, incz;

dgt(n, x, incx, y, incy, lz, incz);
```

Single precision:

```
float x[], y[];
logical lz[];
int n, incx, incy, incz;

sgt(n, x, incx, y, incy, lz, incz);
```

Integer:

```
int x[], y[];
logical lz[];
int n, incx, incy, incz;

igt(n, x, incx, y, incy, lz, incz);
```

C Equivalent

```
for(i = 0; i < n; i++)
      lz[i] = x[i] > y[i];
```

IxAMAX() IxAMAX()

Index of maximum absolute value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
double x[];
int n, incx, i, idamax();

i = idamax(n, x, incx);
```

Single precision:

```
float x[];
int n, incx, i, isamax();

i = isamax(n, x, incx);
```

C Equivalent

```
idamax = -1;
if (n > 0) idamax = 0;
for (i = 1; i < n; i++)
      if (ABS(x[i]) > ABS(x[idamax]) idamax = i;
```

IxAMIN() IxAMIN()

Index of minimum absolute value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
double x[];
int n, incx, i, idamin();

i = idamin(n, x, incx);
```

Single precision:

```
float x[];
int n, incx, i, isamin();

i = isamin(n, x, incx);
```

C Equivalent

```
idamin = 1;
if (n > 0) idamin = 0;
for(i = 1; i < n; i++)
      if (ABS(x[i]) < ABS(x[idamin]) idamin = i;
```

*x*ICLIP() *x*ICLIP()

Inverted clip.

Calling Sequence

Double precision:

```
double x[], alpha, beta, y[];
int n, incx, incy;

diclip(n, x, incx, &alpha, &beta, y, incy);
```

Single precision:

```
float x[], alpha, beta, y[];
int n, incx, incy;

siclip(n, x, incx, &alpha, &beta, y, incy);
```

C Equivalent

```
for(i = 0; i < n; i++) {
   if (x[i] < (alpha + beta)/2.0) {
               y[i] = MIN(x[i], alpha);
   }
   else
               y[i] = MAX(x[i], beta);
}
```

Description

$$y[i] = \begin{cases} \texttt{alpha if alpha < x[i] < (alpha + beta)/2.0} \\ \texttt{beta if (alpha + beta)/2.0} \leq \texttt{x[i] < beta} \\ \texttt{x[i] otherwise} \end{cases}$$

Note that "alpha" must be less than or equal to "beta".

ICOUNT() ICOUNT()

Number of logical true values. (Integer function returning a value.)

Calling Sequence

```
logical lx[];
int n, incx, i, icount();

i = icount(n, lx, incx);
```

C Equivalent

```
icount = 0;
for(i = 0; i < n; i++)
      if (lx[i]) icount++;
```

*x*IFFT() *x*IFFT()

Inverse Fast Fourier Transform.

Calling Sequence

Complex:

```
complex x[], y[];
int n, incx, incy;

cifft(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;

zifft(n, x, incx, y, incy);
```

C Equivalent

$y = fft^{-1}(x)$

Description

This routine performs a one-dimensional inverse **fft** with a maximum vector length of 16384; *n* must be a power of two. The vectors x and y may not overlap because the transform is not done in place.

The **cifft** routine uses **vpfast_** as a scratch vector.

The **zifft** routine dynamically allocates a buffer of $(3n+1)$ eight-byte words the first time you call it. This buffer is reused if subsequent calls to **zfft** use a value of *n* less than or equal to the first, making this routine very fast. If a value of *n* that is larger than the currently allocated buffer is used, the buffer is freed and a larger one is allocated.

If either *incx* or *incy* is not 1, **zifft** will run slower. If both *incx* or *incy* are not 1, **izfft** allocates an additional $2n$ eight-byte words.

IFIRST() IFIRST()

Index of first logical true value. (Integer function returning a value.)

Calling Sequence

```
logical lx[];
int n, incx, i, ifirst();

i = ifirst(n, lx, incx);
```

C Equivalent

```
ifirst = -1;
for(i = 0; i < n; i++) {
      if (lx[i]) {
         ifirst = i;
         break;
      }
}
```

ILAST() ILAST()

Index of last logical true value. (Integer function returning a value.)

Calling Sequence

```
logical lx[];
int n, incx, i, ilast();

i = ilast(n, lx, incx);
```

C Equivalent

```
ilast = -1;
for(i = 0; i < n; i++)
      if (lx[i]) ilast = i;
```

IxMAX() IxMAX()

Index of maximum value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
double x[];
int n, incx, i, idmax();

i = idmax(n, x, incx);
```

Single precision:

```
float x[];
int n, incx, i, ismax();

i = ismax(n, x, incx);
```

C Equivalent

```
idmax = -1;
if (n > 0) idmax = 0;
for(i = 1; i < n; i++)
      if (x[i] > x[idmax]) idmax = i;
```

IxMIN() IxMIN()

Index of minimum value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
double x[];
int n, incx, i, idmin();

i = idmin(n, x, incx);
```

Single precision:

```
float x[];
int n, incx, i, ismin();

i = ismin(n, x, incx);
```

C Equivalent

```
idmin = -1;
if (n > 0) idmin = 0;
for(i = 1; i < n; i++)
      if (x[i] < x(idmin)) idmin = i;
```

LAND() LAND()

Vector logical AND with vector.

Calling Sequence

```
logical x[], y[], z[];
int n, incx, incy, incz;

land(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      z[i] = x[i] && y[i];
```

LANY() LANY()

Logical true if any elements in X are true. (Logical function returning a value.)

Calling Sequence

```
logical x[], l, lany();
int n, incx;

l = lany(n, x, incx);
```

C Equivalent

```
lany = 0;
for(i  = 0; i < n; i++)
      if (x[i]) lany = 1;
```

*x*LBIDI() *x*LBIDI()

Performs the forward substitution to solve the lower bidiagonal in a matrix.

Calling Sequence

Double precision:

```
double l[], b[], x[];
int n, incl, incb, incx;

dlbidi(n, l, incl, b, incb, x, incx);
```

Single precision:

```
float l[], b[], x[];
int n, incl, incb, incx;

slbidi(n, l, incl, b, incb, x, incx);
```

C Equivalent

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

Description

xlbidi does the forward substitution to solve Ax=b where A is the bidiagonal matrix:

```
[  1   0   0   0          ...  0 ]
[ 11   1   0   0          ...  0 ]
[  0  12   1   0          ...  0 ]
[ ... ...                 ...    ]
[ ... ...                 ...    ]
[ ... ...          ln-2    1   0 ]
[ ... ...               ln-1   1 ]
```

l is the lower diagonal (elements 1 through *n*-1 are used)

b is an input (to solve for) (elements 0 through *n*-1 are used)

x is the output (partial answer) (elements 0 through *n*-1 are used)

*x*LBIDI() *(cont.)*

*x*LBIDI() *(cont.)*

If the matrix is a tridiagonal, you can use ***x*trfac()** to perform an LU factorization of **A**, and then use ***x*lbidi()** to solve the lower bidiagonal and ***x*ubidi()** to solve the upper bidiagonal. See ***x*trfac()** and ***x*ubidi()** for more information.

LNOT() LNOT()

Vector logical negation.

Calling Sequence

```
logical x[], y[];
int n, incx, incy;

lnot(n, x, incx, y, incy);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      y[i] = !x[i];
```

LOR() LOR()

Vector logical OR with vector.

Calling Sequence

```
logical x[], y[], z[];
int n, incx, incy, incz;

lor(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      z[i] = x[i] || y[i];
```

LSAND() LSAND()

Scalar logical AND with vector.

Calling Sequence

```
logical x[], y[], alpha;
int n, incx, incy;

lsand (n, &alpha, x, incx, y, incy);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      y[i] = alpha && x[i];
```

LSOR() LSOR()

Scalar logical OR with vector.

Calling Sequence

```
logical x[], y[], alpha;
int n, incx, incy;

lsor(n, &alpha, x, incx, y, incy);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      y[i] = alpha || x[i];
```

*x*MASK() *x*MASK()

Conditional assignment.

Calling Sequence

Double precision:

```
double w[], x[], z[];
logical ly[];
int n, incw, incx, incy, incz;

dmask(n, w, incw, x, incx, ly, incy, z, incz);
```

Single precision:

```
float w[], x[], z[];
logical ly[];
int n, incw, incx, incy, incz;

smask(n, w, incw, x, incx, ly, incy, z, incz);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      if (ly[i])
         z[i] = w[i];
      else
         z[i] = x[i];
```

*x*NE() *x*NE()

Vector element inequality.

Calling Sequence

Double precision:

```
double x[], y[];
logical lz[];
int n, incx, incy, incz;

dne(n, x, incx, y, incy, lz, incz);
```

Single precision:

```
float x[], y[];
logical lz[];
int n, incx, incy, incz;

sne(n, x, incx, y, incy, lz, incz);
```

Integer:

```
int x[], y[];
logical lz[];
int n, incx, incy, incz;

ine(n, x, incx, y, incy, lz, incz);
```

C Equivalent

```
for(i  = 0; i < n; i++)
      lz[i] = x[i] != y[i];
```

*x*NEG() *x*NEG()

Change sign.

Calling Sequence

Double precision:

```
double x[];
int n, incx;

dneg(n, x, incx);
```

Single precision:

```
float x[];
int n, incx;

sneg(n, x, incx);
```

Integer:

```
int x[];
int n, incx;

ineg(n, x, incx);
```

Complex:

```
complex x[];
int n, incx;

cneg(n, x, incx);
```

Double precision complex:

```
zcomplex x[];
int n, incx;

zneg(n, x, incx);
```

*x*VNEG()*(cont.)*

*x*VNEG()*(cont.)*

C Equivalent

```
for(i  = 0; i < n; i++)
      x[i] = -x[i];
```

*x*NRM2() *x*NRM2()

Euclidean vector norm. (Function returning a value.)

Calling Sequence

Double precision:

```
double d, x[], dnrm2();
int n, incx;

d = dnrm2(n, x, incx);
```

Single precision:

```
float s, x[], snrm2();
int n, incx;

s = snrm2(n, x, incx);
```

C Equivalent

```
dnrm2 = 0.0;
for(i  = 0; i < n; i++)
      dnrm2 = dnrm2 + x[i]*x[i];
dnrm2 = sqrt(dnrm2);
```

Description

$$\underline{d}nrm2 = \sqrt{x[0]^2 + x[1]^2 + \ldots + x[n-1]^2}$$

Note that the summation of the squares operation is performed with scaling to protect from overflow or underflow of intermediate results.

*x*RAMP() *x*RAMP()

Ramp function.

Calling Sequence

Double precision:

```
double alpha, beta, x[];
int n, incx;

dramp(n, &alpha, &beta, x, incx);
```

Single precision:

```
float alpha, beta, x[];
int n, incx;

sramp(n, &alpha, &beta, x, incx);
```

Integer:

```
int alpha, beta, x[];
int n, incx;

iramp(n, &alpha, beta, x, incx);
```

C Equivalent

```
for(i = 0; i < n; i++)
      x[i] = alpha + (i - 1) * beta;
```

*x*RANDOM() *x*RANDOM()

Pseudo-random number generation. (Function returning a value.)

Calling Sequence

Double precision:

```
double d, drandom();
d = drandom();
```

Single precision:

```
float s, srandom();
s = srandom();
```

Description

The scalar and vector versions of the random number routines use the same seed value, which is stored in:

```
extern int random_seed_;
```

Since the same seed is used by all the random routines, a call to the vector routine and *n* calls to the scalar routine would produce the same sequence of numbers (i.e., it is possible to vectorize random number calls and expect the same sequence of random numbers).

The seed's initial value is `1` on each node. The user probably will want to set the seed to a different value on each node. Legal values are in the range from `1` through $2^{31}-1$.

The algorithm used is described in the following article:

Park, Stephen K., and Miller, Keith W.
Random Number Generators: Good Ones are Hard to Find.
Communications of the ACM, Vol. 31, Number 10, (Oct 1988), pg. 1192-1207

*x*ROT() *x*ROT()

Apply a plane rotation.

Calling Sequence

Double precision:

```
double x[], y[], c, s;
int n, incx, incy;
drot(n, x, incx, y, incy, &c, &s);
```

Single precision:

```
float x[], y[], c, s;
int n, incx, incy;
srot(n, x, incx, y, incy, &c, &s);
```

C Equivalent

```
for(i = 0; i < 0; i++) {
      t = x[i];
      x[i] = t * c + y[i] * s;
      y[i] = -t * s + y[i] * c;
}
```

Description

Typically, this routine is used to apply a plane rotation to a matrix after using the appropriate *x*rotg() routine to construct a Givens plane rotation. These subroutines compute the new x and y as a function of the sin and cos of the angle through which the matrix is rotated (*c* represents the cosine and *s* represents the sine, the values for which can be obtained with the *x*rotg() routines):

$$x[i]_{new} = x[i]_{old} * c + y[i]_{old} * s$$

$$y[i]_{new} = -x[i]_{old} * s + y[i]_{old} * c$$

*x*ROTG() *x*ROTG()

Construct a Givens plane rotation. It is intended to be used with *x*rot().

Calling Sequence

Double precision:

```
double a, b, c, s;
drotg(&a, &b, &c, &s);
```

Single precision:

```
float a, b, c, s;
srotg(&a, &b, &c, &s);
```

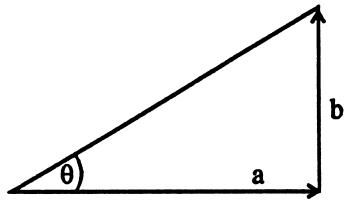
Discussion

Given *a* and *b* as the x and y values, this routine calculates the sine (*s*) and cosine (*c*) of the angle θ through which the vectors are to be rotated. These values can then be used in the *x*rot() routines to apply the plane rotation.

The routines return *r* overwriting *a*, and *z* overwriting *b*, as well as returning *c* and *s* (when you call the routines, *c* and *s* are uninitialized), where:

$r = \sigma\,(a^2 + b^2)^{1/2}$

where $\sigma = (\text{sgn}(a) \text{ if } |a| > |b|) \text{ or } (\text{sgn}(b) \text{ if } |a| \geq |b|)$



z is equal to *s*, 1/c, or 1, depending on the following:

s	if $\|a\| > \|b\|$
1/c	if $\|a\| \geq \|b\|$ and $c \neq 0$
1	if $c = 0$

*x*ROTG() *(cont.)*

*x*ROTG() *(cont.)*

If you wish to reconstruct c and s from z, you can do it as follows:

if $z = 0$	set $c = 0$ and $s = 1$
if $\|z\| < 1$	set $c = (1 - z^2)^{1/2}$ and $s = z$
if $\|z\| > 1$	set $c = 1/z$ and $s = (1 - c^2)^{1/2}$

*x*SADD() *x*SADD()

Scalar plus vector.

Calling Sequence

Double precision:

```
double x[], y[], alpha;
int n, incx, incy;

dsadd(n, &alpha, x, incx, y, incy);
```

Single precision:

```
float x[], y[], alpha;
int n, incx, incy;

ssadd(n, &alpha, x, incx, y, incy);
```

Integer:

```
int x[], y[], alpha;
int n, incx, incy;

isadd(n, &alpha, x, incx, y, incy);
```

Complex:

```
complex x[], y[], alpha;
int n, incx, incy;

csadd(n, &alpha, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[], alpha;
int n, incx, incy;

zsadd(n, &alpha, x, incx, y, incy);
```

*x*SADD() *(cont.)*

*x*SADD() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = alpha + x[i];
```

*x*SCAL() *x*SCAL()

Scalar times a vector to itself.

Calling Sequence

Double precision:

```
double alpha, x[];
int n, incx;

dscal(n, &alpha, x, incx);
```

Single precision:

```
float alpha, x[];
int n, incx;

sscal(n, &alpha, x, incx);
```

Complex:

```
complex alpha, x[];
int n, incx;

cscal(n, &alpha, x, incx);
```

Double precision complex:

```
zcomplex alpha, x[];
int n, incx;

zscal(n, &alpha, x, incx);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      x[i] = alpha * x[i];
```

SCASUM() SCASUM()

Sum of the absolute values of the real and imaginary parts of a complex vector. (Function returning a value.)

Calling Sequence

Single precision:

```
float s, scasum();
complex x[];
int n, incx

s = scasum(n, x, incx)
```

C Equivalent

```
scasum = 0.0;
for(i = 0; i < 0; i++)
      scasum = scasum + ABS(x[i].r) + ABS(x[i].i);
```

Description

$$\text{scasum} = \sum_{i=0}^{n-1} |x[i].r| + |x[i].i|$$

*x*SCATR() *x*SCATR()

Vector scatter.

Calling Sequence

Double precision:

```
double x[], z[];
int n, incx, incy, incz, iy[];

dscatr(n, x, incx, iy, incy, z, incz);
```

Single precision:

```
float x[], z[];
int n, incx, incy, incz, iy[];

sscatr(n, x, incx, iy, incy, z, incz);
```

Integer:

```
integer x[], z[];
int n, incx, incy, incz, iy[];

iscatr(n, x, incx, iy, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[iy[i]] = x[i];
```

Note that *iy* must contain values between `0` and *n*−1. If the values of *iy* are not distinct, the results are undefined.

*x*SDIV() *x*SDIV()

Scalar divided by vector.

Calling Sequence

Double precision:

```
double x[], y[], alpha;
int n, incx, incy;

dsdiv(n, &alpha, x, incx, y, incy);
```

Single precision:

```
float x[], y[], alpha;
int n, incx, incy;

ssdiv(n, &alpha, x, incx, y, incy);
```

Integer:

```
int x[], y[], alpha;
int n, incx, incy;

isdiv(n, &alpha, x, incx, y, incy);
```

Complex:

```
complex x[], y[], alpha;
int n, incx, incy;

csdiv(n, &alpha, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[], alpha;
int n, incx, incy;

zsdiv(n, &alpha, x, incx, y, incy)
```

*x*SDIV() *(cont.)*

*x*SDIV() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = alpha / x[i];
```

*x*SEQ() *x*SEQ()

Vector equal to scalar.

Calling Sequence

Double precision:

```
double x[], alpha;
long ly[];
int n, incx, incy;

dseq(n, &alpha, x, incx, ly, incy) ;
```

Single precision:

```
float x[], alpha;
long ly[];
int n, incx, incy;

sseq(n, &alpha, x, incx, ly, incy);
```

Integer:

```
int x[], alpha;
long ly[];
int n, incx, incy;

iseq(n, &alpha, x, incx, ly, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      ly[i] = alpha == x[i];
```

*x*SGE() *x*SGE()

Scalar greater than or equal to vector.

Calling Sequence

Double precision:

```
double x[], alpha;
long ly[];
int n, incx, incy;

dsge(n, &alpha, x, incx, ly, incy);
```

Single precision:

```
float x[], alpha;
long ly[];
int n, incx, incy;

ssge(n, &alpha, x, incx, ly, incy);
```

Integer:

```
int x[], alpha;
long ly[];
int n, incx, incy;

isge(n, &alpha, x, incx, ly, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      ly[i] = alpha >= x[i];
```

*x*SGT() *x*SGT()

Scalar greater than vector.

Calling Sequence

Double precision:

```
double x[], alpha;
long ly[];
int n, incx, incy;

dsgt(n, &alpha, x, incx, ly, incy);
```

Single precision:

```
float x[], alpha;
long ly[];
int n, incx, incy;

ssgt(n, &alpha, x, incx, ly, incy);
```

Integer:

```
int x[], alpha;
long ly[];
int n, incx, incy;

isgt(n, &alpha, x, incx, ly, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      ly[i] = alpha > x[i];
```

*x*SLE() *x*SLE()

Scalar less than or equal to vector.

Calling Sequence

Double precision:

```
double x[], alpha;
long ly[];
int n, incx, incy;

dsle(n, &alpha, x, incx, ly, incy);
```

Single precision:

```
float x[], alpha;
long ly[];
int n, incx, incy;

ssle(n, &alpha, x, incx, ly, incy);
```

Integer:

```
int x[], alpha;
long ly[];
int n, incx, incy;

isle(n, &alpha, x, incx, ly, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      ly[i] = alpha <= x[i];
```

*x*SLT() *x*SLT()

Scalar less than vector.

Calling Sequence

Double precision:

```
double x[], alpha;
long ly[];
int n, incx, incy;

dslt(n, &alpha, x, incx, ly, incy);
```

Single precision:

```
float x[], alpha;
long ly[];
int n, incx, incy;

sslt(n, &alpha, x, incx, ly, incy);
```

Integer:

```
int x[], alpha;
long ly[];
int n, incx, incy;

islt(n, &alpha, x, incx, ly, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      ly[i] = alpha < x[i];
```

*x*SMUL() *x*SMUL()

Scalar times vector.

Calling Sequence

Double precision:

```
double x[], y[], alpha;
int n, incx, incy;

dsmul(n, &alpha, x, incx, y, incy);
```

Single precision:

```
float x[], y[], alpha;
int n, incx, incy;

ssmul(n, &alpha, x, incx, y, incy);
```

Integer:

```
int x[], y[], alpha;
int n, incx, incy;

ismul(n, &alpha, x, incx, y, incy);
```

Complex:

```
complex x[], y[], alpha;
int n, incx, incy;

csmul(n, &alpha, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[], alpha;
int n, incx, incy;

zsmul(n, &alpha, x, incx, y, incy);
```

*x*SMUL() *(cont.)*

*x*SMUL() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = alpha * x[i];
```

*x*SNE() *x*SNE()

Vector not equal to scalar.

Calling Sequence

Double precision:

```
double x[], alpha;
long ly[];
int n, incx, incy;

dsne(n, &alpha, x, incx, ly, incy);
```

Single precision:

```
float x[], alpha;
long ly[];
int n, incx, incy;

ssne(n, &alpha, x, incx, ly, incy);
```

Integer:

```
int x[], alpha;
long ly[];
int n, incx, incy;

isne(n, &alpha, x, incx, ly, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      ly[i] = alpha != x[i];
```

*x*SOLR() *x*SOLR()

Second-order recurrence routine.

Calling Sequence

Double precision:

```
double w[], x[], y[], z[];
int n, incw, incx, incy, incz;

call dsolr(n, w, incw, x, incx, y, incy, z, incz);
```

Single precision:

```
float w[], x[], y[], z[];
int n, incw, incx, incy, incz;

call ssolr(n, w, incw, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i+2] = w[i] + z[i+1] * x[i] + z[i] * y[i];
```

Note that the routine accesses two more elements in z than in w or x. Be sure to make the z array large enough (including the increment size) to avoid overwriting other memory.

*x*SSUB() *x*SSUB()

Scalar vector subtraction.

Calling Sequence

Double precision:

```
double x[], y[], alpha;
int n, incx, incy;

dssub(n, &alpha, x, incx, y, incy);
```

Single precision:

```
float x[], y[], alpha;
int n, incx, incy;

sssub(n, &alpha, x, incx, y, incy);
```

Integer:

```
int x[], y[], alpha;
int n, incx, incy;

issub(n, &alpha, x, incx, y, incy);
```

Complex:

```
complex x[], y[], alpha;
int n, incx, incy;

cssub(n, &alpha, x, incx, y, incy);
```

Double Precision Complex:

```
zcomplex x[], y[], alpha;
int n, incx, incy;

zssub(n, &alpha, x, incx, y, incy);
```

*x*SSUB() *(cont.)*

*x*SSUB() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = alpha - x[i];
```

*x*SUM() *x*SUM()

Vector sum. (Function returning a value.)

Calling Sequence

Double precision:

```
double d, x[], dsum();
int n, incx;

d = dsum(n, x, incx);
```

Single precision:

```
float s, x[], ssum();
int n, incx;

s = ssum(n, x, incx);
```

Complex:

```
complex c, x[], csum();
int n, incx;

c = csum(n, x, incx);
```

Double precision complex:

```
zcomplex z, x[], zsum();
int n, incx;

z = zsum(n, x, incx);
```

C Equivalent

```
dsum = 0.0;
for(i = 0; i < 0; i++)
      dsum = dsum + x[i];
```

*x*SUM() *(cont.)*

*x*SUM() *(cont.)*

Description

$$\underline{d}sum = \sum_{i=0}^{n-1} x[i]$$

*x*SVMVT() *x*SVMVT()

Scalar minus vector quantity times vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvmvt(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz;

ssvmvt(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = (alpha - x[i]) * y[i];
```

*x*SVPVT() *x*SVPVT()

Scalar plus vector quantity times vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvpvt(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz

ssvpvt(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = (alpha + x[i]) * y[i];
```

*x*SVTSP() *x*SVTSP()

Scalar times vector quantity plus scalar.

Calling Sequence

Double precision:

```
double x[], y[], alpha, beta;
int n, incx, incy;

dsvtsp(n, &alpha, &beta, x, incx, y, incy);
```

Single precision:

```
float x[], y[], alpha, beta;
int n, incx, incy;

ssvtsp(n, &alpha, &beta, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = alpha * x[i] + beta;
```

*x*SVTVM() *x*SVTVM()

Scalar times vector quantity minus vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvtvm(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz;

ssvtvm(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = alpha * x[i] - y[i];
```

*x*SVTVP() *x*SVTVP()

Scalar times vector quantity plus vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvtvp(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha
int n, incx, incy, incz;

ssvtvp(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = alpha * x[i] + y[i];
```

*x*SVVMT() *x*SVVMT()

Scalar times the quantity of vector minus vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvvmt(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz;

ssvvmt(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = alpha * (x[i] - y[i]);
```

*x*SVVPT() *x*SVVPT()

Scalar times the quantity of vector plus vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvvpt(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz;

ssvvpt(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = alpha * (x[i] + y[i]);
```

*x*SVVTM() *x*SVVTM()

Scalar minus the quantity of vector times vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvvtm(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz;

ssvvtm(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = alpha - x[i] * y[i];
```

*x*SVVTP() *x*SVVTP()

Scalar plus the quantity of vector times vector.

Calling Sequence

Double precision:

```
double x[], y[], z[], alpha;
int n, incx, incy, incz;

dsvvtp(n, &alpha, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[], alpha;
int n, incx, incy, incz;

ssvvtp(n, &alpha, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = alpha + x[i] * y[i];
```

*x*SWAP() *x*SWAP()

Swap vectors.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dswap(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

sswap(n, x, incx, y, incy);
```

Integer:

```
int x[], y[];
int n, incx, incy;

iswap(n, x, incx, y, incy);
```

Complex:

```
complex x[], y[];
int n, incx, incy;

cswap(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;

zswap(n, x, incx, y, incy);
```

*x*SWAP() *(cont.)*

*x*SWAP() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++) {
      t = y[i];
      y[i] = x[i];
      x[i] = t;
}
```

*x*TRFAC() *x*TRFAC()

Performs an LU factorization of the tridiagonal of a matrix, and is intended to be used with ***x*ubidi()** and ***x*lbidi()**.

Calling Sequence

Double precision:

```
double l[], d[], u[];
int n, incl, incd, incu;

dtrfac(n, l, incl, d, incd, u, incu);
```

Single precision:

```
float d[], u[], x[];
int n, incd, incu, incx;

strfac(n, l, incl, d, incd, u, incu);
```

C Equivalent

```
d[0] = 1.0/d[0];
for(i=1;i < 0;i++) {
      l[i] = l[i] * d[i-1];
      d[i] = 1.0/(d[i] - l[i] * u[i-1]);
}
```

Discussion

***x*trfac** does an LU factorization of **A**, where **A** is the tridiagonal matrix:

```
[ d1  u1   0   0            ...   0 ]
[ l2  d2  u2   0            ...   0 ]
[ 0   l3  d3  u3            ...   0 ]
[ ... ...                   ...     ]
[ ... ...                   ...     ]
[ ... ...         ln-1 dn-1 un-1    ]
[ ... ...               ln   dn     ]
```

*x*TRFAC() *(cont.)*

*x*TRFAC() *(cont.)*

The matrix is stored in the following space-efficient way:

l is the lower diagonal	(elements 2 through *n* are used)
d is the diagonal	(elements 1 through *n* are used)
u is the upper diagonal	(elements 1 through *n*-1 are used)

***x*trfac()** factors the tridiagonal to produce the lower and upper bidiagonals, which you can then solve with ***x*lbidi()** and ***x*ubidi()**, respectively. In addition, ***x*trfac()** inverts the diagonal, so it is ready for ***x*ubidi()** to solve.

*x*UBIDI() *x*UBIDI()

Performs the backward substitution to solve the tridiagonal in a matrix, and is intended to be used with ***x*trfac()** and ***x*lbidi()**.

Calling Sequence

Double precision:

```
double d[], u[], x[];
int n, incd, incu, incx;

dubidi(n, d, incd, u, incu, x, incx);
```

Single precision:

```
float d[], u[], x[];
int n, incd, incu, incx;

subidi(n, d, incd, u, incu, x, incx);
```

C Equivalent

```
for(i = n - 2; i <- 1; i--)
      x[i] = (x[i] - u[i]) * x[i + 1] * d[i];
```

Discussion

***x*ubidi** does the backward substitution to solve Ax=b where A is the bidiagonal matrix:

```
[  d0   u0    0    0        ... 0 ]
[  0    d1   u1    0        ... 0 ]
[  0     0   d2   u2        ... 0 ]
[ ...   ...                 ...   ]
[ ...   ...                 ...   ]
[ ...   ...           0 dn-2 un-2 ]
[ ...   ...             dn-1 un-1 ]
```

*x*UBIDI() *(cont.)*

*x*UBIDI() *(cont.)*

d is the diagonal that results from	(elements 0 through *n*-1 are used) the use of ***x*trfac()**, which takes the reciprocal of the diagonal
u is the upper diagonal	(elements 0 through *n*-2 are used)
x is an input (the output of ***x*lbidi**) and an	(elements 0 through *n*-1 are used) output (x that satisfies Ax=b)

The ***x*ubidi()** routine does a forward substitution to solve the upper bidiagonal. The ***x*ubidi()** routine assumes that the diagonal is inverted, done automatically if you have a tridiagonal matrix and use ***x*trfac()** to factor it into upper and lower bidiagonals. If your original matrix has only the upper bidiagonal, invert the diagonal before you use ***x*ubidi()**.

*x*VABS() *x*VABS()

Element-wise absolute value.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvabs (n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svabs(n, x, incx, y, incy);
```

Integer:

```
int x[], y[];
int n, incx, incy;

ivabs(n, x, incx, y, incy)
```

Complex:

```
complex x[];
float sy[];
int n, incx, incy;

cvabs(n, x, incx, sy, incy);
```

Double Precision Complex:

```
zcomplex x[];
double dy[];
int n, incx, incy;

zvabs(n, x, incx, dy, incy);
```

*x*VABS() *(cont.)*

*x*VABS() *(cont.)*

C Equivalent

For non-complex values:

```
for(i = 0; i < 0; i++)
        y[i] = ABS(x[i]);
```

For complex values:

```
for(i = 0; i < 0; i++)
        y[i] = CABS(x[i]);
```

Description

y[i] = | x[i] |

*x*VADD() *x*VADD()

Vector addition.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvadd(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svadd(n, x, incx, y, incy, z, incz);
```

Integer:

```
int x[], y[], z[];
int n, incx, incy, incz;

ivadd(n, x, incx, y, incy, z, incz);
```

Complex:

```
complex x[], y[], z[];
int n, incx, incy, incz;

cvadd(n, x, incx, y, incy, z, incz);
```

Double precision complex:

```
zcomplex x[], y[], z[];
int n, incx, incy, incz;

zvadd(n, x, incx, y, incy, z, incz);
```

*x*VADD() *(cont.)*

*x*VADD() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = x[i] + y[i];
```

*x*VAMAX() *x*VAMAX()

Vector element-wise maximum absolute value.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvamax(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svamax(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = MAX(ABS(x[i]),ABS(y[i]));
```

Description

z[i] = max (| x[i] |, | y[i] |)

*x*VAMIN() *x*VAMIN()

Vector element-wise minimum absolute value.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvamin(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svamin(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = MIN(ABS(x[i]),ABS(y[i]));
```

Description

z[i] = min (| x[i] |, | y[i] |)

*x*VATAN() *x*VATAN()

Element-wise inverse-tangent.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvatan(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svatan(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = atan(x[i]);
```

Description

y[i] = arctangent (in radians) of x[i]

*x*VATN2() *x*VATN2()

Element-wise inverse-tangent.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvatn2(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svatn2(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = atan2(x[i],y[i]);
```

Description

z[i] = arctangent (in radians) of x[i] / y[i]

*x*VCMPLX() *x*VCMPLX()

Construction of a complex vector. The **cvcmplx()** routine builds a complex vector from two float vectors; the **zvcmplx()** routine builds a double precision complex vector from two double vectors.

Calling Sequence

Float to complex:

```
complex z[];
float sx[], sy[];
int n, incx, incy, incz;

cvcmplx(n, sx, incx, sy, incy, z, incz);
```

Double to double precision complex:

```
zcomplex z[];
double dx[], dy[];
int n, incx, incz;

zvcmplx(n, dx, incx, dy, incy, z, incz);
```

C Equivalent

Float to complex:

```
for(i = 0; i < 0; i++)
      z[i] = CMPLX(sx[i], sy[i]);
```

Double to double precision complex:

```
for(i = 0; i < 0; i++)
      z[i] = CMPLX(dx[i], dy[i]);
```

Description

Construction of a complex vector from two float vectors, the first *x* being the real part, and the second *y* being the imaginary part.

*x*VCONJG() *x*VCONJG()

Calculate the conjugate of a complex vector.

Calling Sequence

Complex:

```
complex x[], y[];
int n, incx, incy;

call cvconjg(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;

call zvconjg(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = CONJG(x[i]);
```

*x*VCOS() *x*VCOS()

Element-wise cosine.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvcos(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svcos(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = cos(x[i]);
```

Description

y[i] = cosine of x[i] in radians

VDBLE() VDBLE()

Single to double precision.

Calling Sequence

```
double dy[];
float sx[];
int n, incx, incy;

vdble(n, sx, incx, dy, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      dy[i] = (double)sx[i];
```

Description

dy[i] = conversion of sx[i] to double precision

*x*VDIV() *x*VDIV()

Element-wise vector division.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvdiv (n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svdiv(n, x, incx, y, incy, z, incz);
```

Integer:

```
int x[], y[], z[];
int n, incx, incy, incz;

ivdiv(n, x, incx, y, incy, z, incz);
```

Complex:

```
complex x[], y[], z[];
int n, incx, incy, incz;

cvdiv(n, x, incx, y, incy, z, incz);
```

Double precision complex:

```
zcomplex x[], y[], z[];
int n, incx, incy, incz

zvdiv(n, x, incx, y, incy, z, incz);
```

*x*VDIV() *(cont.)*

*x*VDIV() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = x[i] / y[i];
```

*x*VEXP() *x*VEXP()

Element-wise exponential.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvexp(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy

svexp(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = exp(x[i]);
```

Description

$y[i] = e^{x[i]}$

*x*VFIX() *x*VFIX()

Truncate elements to integer values.

Calling Sequence

Double precision:

```
double x[];
int n, iy[], incx, incy;

dvfix(n, x, incx, iy, incy);
```

Single precision:

```
float x[];
int n, iy[], incx, incy;

svfix(n, x, incx, iy, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      iy[i] = (int)x[i];
```

Description

iy[i] = value resulting from truncation of the fractional part of x[i]

*x*VFLOA() *x*VFLOA()

Convert integer to floating point.

Calling Sequence

Double precision:

```
double y[];
int n, incx, incy, ix[];

dvfloa(n, ix, incx, y, incy);
```

Single precision:

```
float y[];
int n, incx, incy, ix[];

svfloa(n, ix, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = ix[i];
```

Description

y[i] = conversion of ix[i] to floating point

*x*VIMAG() *x*VIMAG()

Extract the imaginary part of a complex vector.

Calling Sequence

Complex:

```
complex x[];
float sy[];
int n, incx, incy;

cvimag(n, x, incx, sy, incy);
```

Double precision complex:

```
zcomplex x[];
double dy[];
int n, incx, incy;

zvimag(n, x, incx, dy, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      sy[i] = x[i].i;
```

*x*VLG10() *x*VLG10()

Element-wise base 10 logarithm.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvlg10(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svlg10(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < n; i++)
      y[i] = log10(x[i]);
```

Description

$y[i] = \log_{10}(x[i])$

*x*VLOG() *x*VLOG()

Element-wise natural logarithm.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvlog(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svlog (n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = log(x[i]);
```

Description

y[i] = ln (x[i])

*x*VMAX() *x*VMAX()

Vector element-wise maximum.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvmax(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

dvmax(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = MAX(x[i],y[i]);
```

*x*VMIN() *x*VMIN()

Vector element-wise minimum.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvmin(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svmin(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = MIN(x[i],y[i]);
```

*x*VMUL() *x*VMUL()

Element-wise vector multiplication.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;

dvmul(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz;

svmul(n, x, incx, y, incy, z, incz);
```

Integer:

```
int x[], y[], z[];
int n, incx, incy, incz;

ivmul(n, x, incx, y, incy, z, incz);
```

Complex:

```
complex x[], y[], z[];
int n, incx, incy, incz;

cvmul(n, x, incx, y, incy, z, incz);
```

Double precision complex:

```
zcomplex x[], y[], z[];
int n, incx, incy, incz;

zvmul(n, x, incx, y, incy, z, incz);
```

*x*VMUL() *(cont.)*

*x*VMUL() *(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = x[i] * y[i];
```

*x*VNEG() *x*VNEG()

Negate vector.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvneg(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svneg(n, x, incx, y, incy);
```

Integer:

```
int x[], y[];
int n, incx, incy;

ivneg(n, x, incx, y, incy);
```

Complex:

```
complex x[], y[];
int n, incx, incy;

cvneg(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;

zvneg(n, x, incx, y, incy);
```

*x*VNEG()*(cont.)*

*x*VNEG()*(cont.)*

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = -x[i];
```

*x*VPOLY() *x*VPOLY()

Vector polynomial evaluation.

Calling Sequence

Double precision:

```
double x[], c[], y[];
int n, m, incx, incc, incy;

dvpoly(n, x, incx, m, c, incc, y, incy);
```

Single precision :

```
float x[], c[], y[];
int n, m, incx, incc, incy;

svpoly(n, x, incx, m, c, incc, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] >= c[m - 1];
for(j = m - 2; j = 0;j--) {
      for(i = 0;i < 0;i++) {
         y[i] = x[i] * y[i] + c[j];
      }
}
```

Description

c is a vector of length 'm' containing the coefficients of the polynomial.

Vector polynomial evaluation;

$$y[i] = c_0 + c_1x[i] + c_2x[i]^2 ... + c_{m-1}x[i]^{m-1}$$

Note that if $m \leq 0$, then $y[i] = 0.0$ for $i = 0, 1, ..., n-1$

*x*VPOW() *x*VPOW()

Element-wise power function.

Calling Sequence

Double precision:

```
double precision x[], y[], z[];
int n, incx, incy, incz;

dvpow(n, x, incx, y, incy, z, incz);
```

Single precision:

```
float x[], y[], z[];
int n, incx, incy, incz

svpow(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = pow(x[i], y[i])
```

Description

$z[i] = x[i]^{y[i]}$

Note that exponentiation is performed using logarithms. The value of *x[i]* cannot be a negative number since logarithms of negative values are undefined. Therefore, *x[i]* must be greater than or equal to zero. Also, if *x[i]* equals zero, then *y[i]* must be greater than zero to avoid a divide by zero.

*x*VRANDOM() *x*VRANDOM()

Pseudo-random vector generator.

Calling Sequence

Double precision:

```
double x[];
int n, incx;

dvrandom(n, x, incx);
```

Single precision :

```
float x[];
int n, incx;

svrandom(n, x, incx);
```

Description

The scalar and vector versions of the random number routines use the same seed value stored in:

```
extern int random_seed_;
```

Since the same seed is used by all the random routines, a call to the vector routine and *n* calls to the scalar routine would produce the same sequence of numbers (i.e., it is possible to vectorize random number calls and expect the same sequence of random numbers).

The seed's initial value is `1` on each node. The user probably will want to set the seed to a different value on each node. Legal values are in the range from `1` through $2^{31}-1$.

The algorithm used is described in the following article:

Park, Stephen K., and Miller, Keith W.
Random Number Generators: Good Ones are Hard to Find.
Communications of the ACM, Vol. 31, Number 10, (Oct 1988), pg. 1192-1207

*x*VREAL() *x*VREAL()

Extract the real part of a complex vector.

Calling Sequence

Complex:

```
complex x[];
float sy[];
integer n, incx, incy;

cvreal(n, x, incx, sy, incy);
```

Double precision complex:

```
zcomplex x[];
double dy[];
integer n, incx, incy;

zvreal(n, x, incx, dy, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      sy[i] = x[i].r;
```

*x*VRECP() *x*VRECP()

Vector reciprocal.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

dvrecp(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

svrecp(n, x, incx, y, incy);
```

Complex:

```
complex x[], y[];
int n, incx, incy;

cvrecp(n, x, incx, y, incy);
```

Double precision complex:

```
zcomplex x[], y[];
int n, incx, incy;

zvrecp(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = 1.0 / x[i];
```

*x*VSIN() *x*VSIN()

Element-wise sine.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

call dvsin(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

call svsin(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0;i < 0;i++)
      y[i] = sin(x[i]);
```

Description

y[i] = sine of x[i] in radians

VSNGL() VSNGL()

Double to single precision.

Calling Sequence

```
double dx[];
int n, incx, incy;
float sy[];

vsngl(n, dx, incx, sy, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      sy[i] = (float)dx[i];
```

Description

sy[i] = conversion of dx[i] to type **float**.

*x*VSQRT() *x*VSQRT()

Element-wise square root.

Calling Sequence

Double precision:

```
double x[], y[];
int n, incx, incy;

call dvsqrt(n, x, incx, y, incy);
```

Single precision:

```
float x[], y[];
int n, incx, incy;

call svsqrt(n, x, incx, y, incy);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      y[i] = sqrt(x[i]);
```

Description

$$y[i] = \sqrt{x[i]}$$

*x*VSUB() *x*VSUB()

Vector subtraction.

Calling Sequence

Double precision:

```
double x[], y[], z[];
int n, incx, incy, incz;
dvsub(n, x, incx, y, incy, z, incz);
Single precision:
float x[], y[], z[];
int n, incx, incy, incz;

svsub(n, x, incx, y, incy, z, incz);
```

Integer:

```
int x[], y[], z[];
int n, incx, incy, incz;

ivsub(n, x, incx, y, incy, z, incz);
```

Complex:

```
complex x[], y[], z[];
int n, incx, incy, incz;

cvsub(n, x, incx, y, incy, z, incz);
```

Double precision complex:

```
zcomplex x[], y[], z[];
int n, incx, incy, incz;

zvsub(n, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = x[i] - y[i];
```

*x*VVMVT() *x*VVMVT()

Vector minus vector quantity times vector.

Calling Sequence

Double precision:

```
double w[], x[], y[], z[];
int n, incw, incx, incy, incz;

dvvmvt(n, w, incw, x, incx, y, incy, z, incz);
```

Single precision:

```
float w[], x[], y[], z[];
int n, incw, incx, incy, incz;

svvmvt(n, w, incw, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = (w[i] - x[i]) * y[i];
```

*x*VVPVT() *x*VVPVT()

Vector plus vector quantity times vector.

Calling Sequence

Double precision:

```
double w[], x[], y[], z[];
int n, incw, incx, incy, incz;

dvvpvt(n, w, incw, x, incx, y, incy, z, incz);
```

Single precision:

```
float w[],x[], y[], z[];
int n, incw, incx, incy, incz;

svvpvt(n, w, incw, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = (w[i] + x[i]) * y[i];
```

*x*VVTVM() *x*VVTVM()

Vector times vector quantity minus vector.

Calling Sequence

Double precision:

```
double w[], x[], y[], z[];
int n, incw, incx, incy, incz;

dvvtvm(n, w, incw, x, incx, y, incy, z, incz);
```

Single precision:

```
float w[], x[], y[], z[];
int n, incw, incx, incy, incz;

svvtvm(n, w, incw, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = w[i] * x[i] - y[i];
```

*x*VVTVP() *x*VVTVP()

Vector times vector quantity plus vector.

Calling Sequence

Double precision:

```
double w[], x[], y[], z[];
int n, incw, incx, incy, incz;

dvvtvp(n, w, incw, x, incx, y, incy, z, incz);
```

Single precision:

```
float w[], x[], y[], z[];
int n, incw, incx, incy, incz;

svvtvp(n, w, incw, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = w[i] * x[i] + y[i];
```

*x*VVVTM() *x*VVVTM()

Vector minus the quantity of vector times vector.

Calling Sequence

Double precision:

```
double precision w[], x[], y[], z[];
int n, incw, incx, incy, incz;

dvvvtm(n, w, incw, x, incx, y, incy, z, incz);
```

Single precision:

```
float w[], x[], y[], z[];
int n, incw, incx, incy, incz;

svvvtm(n, w, incw, x, incx, y, incy, z, incz);
```

C Equivalent

```
for(i = 0; i < 0; i++)
      z[i] = w[i] - x[i] * y[i];
```

FORTRAN VECLIB ROUTINES 2

INTRODUCTION

The Vector Library (VecLib) contains routines that support Intel's numeric products. These routines provide a basic set of vector operations that can be used to replace Fortran DO loops. There are six versions of the VecLib routines contained in the following libraries:

Library	Implemented With	Function
/usr/lib/libvec.a	Fortran and C	Executes on standard node board or system resource manager.
/usr/lib/libsxvec.a	Fortran and C	Executes on SX option board.
/usr/lib/libvxvec.a	Microcode	Executes on vector processor board.
/usr/lib/libvxdbvec.a	Microcode	Executes on vector processor board. Provides debug checks.
/usr/lib/libvxsxvec.a	Microcode	Executes on vector processor board and nodes that have SX option.
/usr/lib/libvxsxdbvec.a	Microcode	Executes on vector processor board and nodes that have SX option. Provides debug checks.

The */usr/lib/libsxvec.a* library can be used only if the SX option is installed in your system. The */usr/lib/libvxsxvec.a* and */usr/lib/libvxsxdbvec.a* libraries can be used only if both the SX option and vector processor boards are installed. The last four libraries (those with *vx* in the names) can be used only if vector processor boards are installed. (Refer to the *iPSC®/2 VX User's Guide* for more information.)

This chapter contains the following information:

- Conventions used in naming routines.
- Conventions used for parameter names in the reference pages (formal parameters).
- General rules and information that will help you use the VecLib routines.
- General limitations.
- Error messages.
- A table that is a summary of all of the routines, listing a brief description of each routine along with a synopsis of the call and the Fortran equivalent of the call.
- The final section contains a reference page for each VecLib routine, in order alphabetically by base name. For each routine, the reference contains a brief definition, the calling sequence, the Fortran equivalent, and a mathematical description.

NAMING CONVENTIONS

Routines exist for both single precision and double precision functions. In some cases, routines exist for integer, logical, and complex functions. The data type a function operates on is indicated by the first or second letter of the routine name; "D" for double, "S" for single, "I" for integer, "L" for logical, and "C" for complex, and "Z" for double precision complex. For example:

Base	Double Precision	Single Precision	Integer	Complex
*x*COPY	DCOPY	SCOPY	ICOPY	CCOPY

The vector library includes most of the Basic Linear Algebra Subprograms (BLAS) and uses the BLAS calling sequence and naming conventions.

NOTE

For more information on the Basic Linear Algebra Subprograms, refer to the following publications:

Bunch, J.R., Dongarra, J.J., Moler, C.B., Stewart, G.W., *LINPACK User's Guide*, 1979, Appendix A.

Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T., "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Software*, 1979, Vol. 5, No. 3, pp. 308-371.

Some of the vector routines use the Reverse Polish Notation (RPN) naming convention. An example is shown in Table 2-1.

Table 2-1. Reverse Polish Notation

Routine	Notation	Description
*x*SVTVM	S = Scalar V = Vector T = Times M = Minus	Scalar times vector quantity minus vector
*x*SVVTP	S = Scalar V = Vector T = Times P = Plus	Scalar plus the quantity of vector times vector

FORMAL PARAMETERS

The calling sequence for each routine always includes the number of iterations as the first parameter, followed by any scalar and vector values. Vector arguments in the parameter list are in **bold** type style to distinguish them from scalar arguments. Each vector argument is followed immediately in the argument list by its increment or "stride" (i.e., **Z**, INCZ). Notice that some routines have an output vector while other routines overwrite one of the input vectors.

Routines which take one vector as input use "**X**." Routines which use two vectors use "**X**" and "**Y**." Routines which use three vectors use "**X**", "**Y**", and "**Z**." Routines which use four vectors use "**W**", "**X**", "**Y**", and "**Z**."

Routines that use one scalar use "ALPHA". Routines that use two scalars use "ALPHA" and "BETA".

In general, formal parameters can be described as follows:

N	number of vector elements to process
X	vector with at least (N-1) * ABS(INCX)+1 elements
INCX	stride of the vector **X**
ALPHA	scalar constant
BETA	scalar constant

GENERAL RULES

- All floating point arguments and results normally match the *type* (for example, double or single) of the subprogram.
- Each routine, as documented in this chapter, includes three parameters:
  - the number of iterations
  - vector and scalar values, if appropriate
  - strides
- In the calling sequence, if the vector *type* differs from the *type* indicated by the routine name, it is indicated by a single letter preceded by an "S" for single, "D" for double, "I" for integer, "L" for logical, a "C" for complex, or a "Z" type vectors. For example, the vector "**LZ**" in the call below is of the *logical* type; "**X**" and "**Y**" are *single precision.*

```
CALL SGT (N, X, INCX, Y, INCY, LZ, INCZ)
```

- Data types follow ANSI Fortran standards. All integer references are INTEGER*4, all complex references are COMPLEX*8, and all logical references are LOGICAL*4.
- The trigonometric routines (*x*VATN2, *x*COS, *x*SIN) expect their arguments to be in radians, or return a vector whose values are expressed in radians.
- Limitations specified in some routines are not checked at runtime. If violated, erroneous results are produced. The only routines which provide some parameter-checking are in *libvxdbvec.a* or *libvxsxdbvec.a.*
- Unless otherwise noted, vector elements which are not contiguous, but are separated by equal distances (either positive or negative), can be accessed by using a stride other than one. Non-unit strides follow the BLAS parameter-passing convention which does not allow negative indices to arrays.

  The formula for indexing through the vectors [*n*,*x*(*lowest$index*),*incx*] for *i* from 1 to '*n*' is:

```
if (incx  0) then
     x((i-1) * incx + lowest$index)
 else
     x((i-n) * incx + lowest$index)
     endif
```

The following example defines two arrays, *x* and *y*. Then, there are three calls to **saxpy** (a routine that multiplies a scalar times a vector, adds the result to another vector, then overwrites the second vector with the result). These calls specify positive and negative strides not equal to 1. For each, the results in *y* are shown.

```
real x (10)
real y (10)
data x /1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0/
data y /.01,.02,.03,.04,.05,.06,.07,.08,.09,.10/
call saxpy (3,1.0,x,2,y,1)
```

y will be equal to (1.01, 3.02, 5.03, .04, .05, .06, .07, .08, .09, .10)

```
call saxpy (3,1.0,x,-2,y,1)
```

y will be equal to (5.01, 3.02, 1.03, .04, .05, .06, .07, .08, .09, .10)

```
call saxpy (3,1.0,x(3),-2,y(2),2)
```

y will be equal to (.01, 7.02, .03, 5.04, .05, 3.06, .07, .08, .09, .10)

GENERAL LIMITATIONS

The following limitations apply to vector routines described in this chapter.

- Routines do nothing for *n* less than or equal to zero.
- Output vector strides of zero produce undefined results if *n* is greater than 1.
- Care must be used when overlapping input and output vectors. Because of the pipelined nature of the VX arithmetic, the results for one iteration (*i*) are not always written to memory before the next iteration (*i*+1) is begun.
- Fortran and C versions of libraries may not produce exactly the same results as microcode versions.
  - 387™ coprocessor hardware supports denormals, VX and SX hardware does not.
  - VX hardware does not support the IEEE divide or square root operations.
  - 387, VX, and SX coprocessors use different algorithms for transcendental functions.
  - Because of the pipelined nature of the VX arithmetic hardware, the accumulation of round-off error for reduction operations is not the same as with the 387 or SX coprocessor.

ERROR MESSAGES

Error messages are generated only when running VecLib routines in *libvxdbvec.a* or *libvxsxdbvec.a*. In these messages, "<x>" represents vector or scalar parameter names and "<ddot>" is used to represent any of the VecLib routine names.

`VX veclib: Error in length for argument <x> in call to <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support

`VX veclib: Error in memory space for argument <x> in call to <ddot>`

Description: The data is not on vector board memory but is on the node board.
Cause: Incorrect use of *vxld*.
Action: Move the data containing the vector to the VX memory using *vxld*.

`VX veclib: Error in physical alignment for argument <x> in call to <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support

`VX veclib: Error in physical bounds check for argument <x> in call to <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support

`VX veclib: Error in type for check of argument <x> in call from <ddot>`

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support

`VX veclib: Error in virtual alignment for argument <x> in call to <ddot>`

Description: The virtual address modulo the data size (in bytes) is not equal to zero.
Cause: Ignored warning from compiler about equivalence misalignment or compiler or linker didn't align the data correctly.
Action: If warning was ignored, change equivalence statement. If compiler or linker didn't align data correctly, call iSC Customer Support.

`VX veclib: Error in virtual bounds check for argument <x> in call to <ddot>`

Description: The data is not contiguous in virtual memory.
Cause: Requesting vector operations outside the bounds of an array.
Action: Locate error in your program.

```
VX veclib:  The following enabled exceptions have occurred: NaN (not-a-number),
            overflow, underflow.
```

Cause: Requested modification of exceptions using *vpexcept* and algorithm-produced exception.

Action: Modify algorithm used so exception does not occur.

*x*ASUM() *x*ASUM()

Sum of the absolute values. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), DASUM
INTEGER*4 N, INCX

D = DASUM(N, X, INCX)
```

Single precision:

```
REAL S, X(*), SASUM
INTEGER*4 N, INCX

S = SASUM(N, X, INCX)
```

Fortran Equivalent

```
      DASUM = 0.0
      DO 10 I = 1, N
         DASUM = DASUM + ABS(X(I))
10    CONTINUE
```

Description

$$\underline{D}ASUM = \sum_{I=1}^{N} |X(I)|$$

*x*AXPY() *x*AXPY()

Scalar times a vector plus a vector to itself.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA, Y(*)
INTEGER*4 N, INCX, INCY

CALL DAXPY(N,ALPHA,X,INCX,Y,INCY)
```

Single precision:

```
REAL X(*), ALPHA, Y(*)
INTEGER*4 N, INCX, INCY

CALL SAXPY(N,ALPHA,X,INCX,Y,INCY)
```

Complex:

```
COMPLEX X(*), ALPHA, Y(*)
INTEGER*4 N, INCX, INCY

CALL CAXPY(N,ALPHA,X,INCX,Y,INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), ALPHA, Y(*)
INTEGER*4 N, INCX, INCY

CALL ZAXPY(N,ALPHA,X,INCX,Y,INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA * X(I) + Y(I)
10    CONTINUE
```

*x*CLIP() *x*CLIP()

Clip to interval (ALPHA, BETA).

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA, Y(*), BETA
INTEGER*4 N, INCX, INCY

CALL DCLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

Single precision:

```
REAL X(*), ALPHA, Y(*), BETA
INTEGER*4 N, INCX, INCY

CALL SCLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = MIN (MAX(X(I), ALPHA), BETA)
10    CONTINUE
```

Description

$$y(i) = \begin{cases} \text{ALPHA IF X(I) < ALPHA} \\ \text{X(I) IF ALPHA £ X(I) £ BETA} \\ \text{BETA IF X(I) > BETA} \end{cases}$$

Note that "ALPHA" must be less than or equal to "BETA".

*x*CNDST() *x*CNDST()

Conditional assignment.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DCNDST(N, X, INCX, LY, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SCNDST (N, X, INCX, LY, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         IF (LY(I)) Z(I) = X(I)
10    CONTINUE
```

*x*COPY() *x*COPY()

Copy vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DCOPY(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SCOPY(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ICOPY(N, X, INCX, Y, INCY)
```

Logical:

```
LOGICAL*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL LCOPY(N, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CCOPY(N, X, INCX, Y, INCY)
```

*x*COPY() *(cont.)*

*x*COPY() *(cont.)*

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZCOPY(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = X(I)
10    CONTINUE
```

*x*DOT() *x*DOT()

Dot product of two vectors. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), Y(*), DDOT
INTEGER*4 N, INCX, INCY

D = DDOT(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL S, X(*), Y(*), SDOT
INTEGER*4 N, INCX, INCY

S = SDOT(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DDOT = 0.0
      DO 10 I = 1, N
         DDOT = DDOT + X(I)*Y(I)
10    CONTINUE
```

Description

$$\underline{D}DOT = \sum_{I=1}^{N} (X(I) \cdot Y(I))$$

*x*DOTC() *x*DOTC()

Dot product of two complex vectors, with the first vector being conjugated. (Function returning a value.)

Calling Sequence

Complex:

```
COMPLEX C, X(*), Y(*), CDOTC
INTEGER*4 N, INCX, INCY

C = CDOTC(N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 Z, X(*), Y(*), ZDOTC
INTEGER*4 N, INCX, INCY

Z = ZDOTC(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      CDOTC = (0.0,0.0)
      DO 10 I = 1, N
         CDOTC = CDOTC + CONJG(X(I)) * Y(I)
10    CONTINUE
```

Description

$$\underline{C}DOTC = \sum_{I=1}^{N} (CONJG(X(I)) \cdot Y(I))$$

*x*DOTU() *x*DOTU()

Dot product of two complex vectors. (Function returning a value.)

Calling Sequence

Complex:

```
COMPLEX C, X(*), Y(*), CDOTU
INTEGER*4 N, INCX, INCY

C = CDOTU(N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 Z, X(*), Y(*), ZDOTU
INTEGER*4 N, INCX, INCY

Z = ZDOTU(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      CDOTU = (0.0,0.0)
      DO 10 I = 1, N
         CDOTU = CDOTU + X(I) * Y(I)
10    CONTINUE
```

Description

$$\underline{C}DOTU = \sum_{I=1}^{N} X(I) \cdot Y(I)$$

DZASUM() DZASUM()

Sum of the absolute values of the real and imaginary parts of a complex vector. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, DZASUM
COMPLEX*16 X(*)
INTEGER*4 N, INCX

D = DZASUM(N, X, INCX)
```

Fortran Equivalent

```
      DZASUM = 0.0
      DO 10 I = 1, N
         DZASUM = DZASUM + ABS(REAL(X(I))) + ABS(IMAG(X(I)))
10    CONTINUE
```

Description

$$DZASUM = \sum_{I=1}^{N} |REAL(X(I))| + |IMAG(X(I))|$$

*x*EQ() *x*EQ()

Vector element equality.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DEQ(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SEQ(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IEQ(N, X, INCX, Y, INCY, LZ, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LZ(I) = X(I) .EQ. Y(I)
10    CONTINUE
```

*x*FFT() *x*FFT()

Fast Fourier Transform.

Calling Sequence

Complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CFFT (N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZFFT (N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
Y = FFT (X)
```

Description

This routine performs a one-dimensional FFT with a maximum vector length of 16384; *N* must be a power of two. The vectors X and Y may not overlap because the transform is not done in place.

The **CFFT** routine uses **VPFAST_** as a scratch vector.

The **ZFFT** routine dynamically allocates a buffer of (3*N* + 1) eight-byte words the first time you call it. This buffer is reused if subsequent calls to **ZFFT** use a value of *N* less than or equal to the first, making this routine very fast. If a value of *N* that is larger than the currently allocated buffer is used, the buffer is freed and a larger one is allocated.

If either *INCX* or *INCY* is not 1, **ZFFT** will run slower. If both *INCX* and *INCY* are not 1, **ZFFT** allocates an additional 2*N* eight-byte words.

*x*FILL() *x*FILL()

Fill vector with scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION ALPHA, X(*)
INTEGER*4 N, INCX

CALL DFILL(N, ALPHA, X, INCX)
```

Single precision:

```
REAL ALPHA, X(*)
INTEGER*4 N, INCX

CALL SFILL(N, ALPHA, X, INCX)
```

Integer:

```
INTEGER*4 ALPHA, X(*)
INTEGER*4 N, INCX

CALL IFILL(N, ALPHA, X, INCX)
```

Logical:

```
LOGICAL*4 ALPHA, X(*)
INTEGER*4 N, INCX

CALL LFILL(N, ALPHA, X, INCX)
```

Complex:

```
COMPLEX ALPHA, X(*)
INTEGER*4 N, INCX

CALL CFILL(N, ALPHA, X, INCX)
```

*x*FILL() *(cont.)*

*x*FILL() *(cont.)*

Double precision complex:

```
COMPLEX*16 ALPHA, X(*)
INTEGER*4 N, INCX

CALL ZFILL(N, ALPHA, X, INCX
```

Fortran Equivalent

```
      DO 10 I = 1, N
         X(I) = ALPHA
10    CONTINUE
```

*x*FOLR() *x*FOLR()

First-order linear recurrence routine.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DFOLR(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SFOLR(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I+1) = Y(I) + Z(I) * X(I)
10    CONTINUE
```

Note that the routine accesses one more element in *Z* than in *W* or *X*. Be sure to make the *Z* array large enough (including the increment size) to avoid overwriting other memory.

*x*GATHR() *x*GATHR()

Vector gather.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)

CALL DGATHR(N, X, INCX, IY, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)

CALL SGATHR(N, X, INCX, IY, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)

CALL IGATHR(N, X, INCX, IY, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(IY(I))
10    CONTINUE
```

Description

The array *IY* must contain values between 1 and *N*.

*x*GE() *x*GE()

Vector element greater than or equal to vector element.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DGE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SGE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IGE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LZ(I) = X(I) .GE. Y(I)
10    CONTINUE
```

*x*GT() *x*GT()

Vector element greater than vector element.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DGT(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SGT(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IGT(N, X, INCX, Y, INCY, LZ, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LZ(I) = X(I) .GT. Y(I)
10    CONTINUE
```

IxAMAX() IxAMAX()

Index of maximum absolute value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDAMAX

I = IDAMAX(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISAMAX

I = ISAMAX(N, X, INCX)
```

Fortran Equivalent

```
      IDAMAX = 0
      IF (N .GT. 0) IDAMAX = 1
      DO 10 I = 2, N
         IF (ABS(X(I)) .GT. ABS(X(IDAMAX))) IDAMAX = I
10    CONTINUE
```

IxAMIN() IxAMIN()

Index of minimum absolute value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDAMIN

I = IDAMIN(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISAMIN

I = ISAMIN(N, X, INCX)
```

Fortran Equivalent

```
      IDAMIN = 0
      IF (N .GT. 0) IDAMIN = 1
      DO 10 I= 2, N
         IF (ABS(X(I)) .LT. ABS(X(IDAMIN))) IDAMIN = I
10    CONTINUE
```

*x*ICLIP() *x*ICLIP()

Inverted clip.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA, BETA, Y(*)
INTEGER*4 N, INCX, INCY

CALL DICLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

Single precision:

```
REAL X(*), ALPHA, BETA, Y(*)
INTEGER*4 N, INCX, INCY

CALL SICLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

Fortran Equivalent

```
DO 10 I = 1, N
      IF (X(I) .LT. (ALPHA + BETA)/2.0) THEN
         Y(I) = MIN(X(I), ALPHA)
      ELSE
         Y(I) = MAX(X(I), BETA)
      ENDIF
10    CONTINUE
```

Description

$$Y(I) = \begin{cases} \text{ALPHA IF ALPHA} < X(I) < (\text{ALPHA} + \text{BETA})/2.0 \\ \text{BETA IF } (\text{ALPHA} + \text{BETA})/2.0 \text{ £ } X(I) < \text{BETA} \\ X(I) \text{ OTHERWISE} \end{cases}$$

Note that "ALPHA" must be less than or equal to "BETA".

ICOUNT() ICOUNT()

Number of logical true values. (Integer function returning a value.)

Calling Sequence

```
LOGICAL*4 LX (*)
INTEGER*4 N, INCX, I, ICOUNT

I = ICOUNT (N, LX, INCX)
```

Fortran Equivalent

```
      ICOUNT = 0
      DO 10 I = 1, N
         IF (LX(I)) ICOUNT = ICOUNT + 1
10    CONTINUE
```

*x*IFFT() *x*IFFT()

Inverse Fast Fourier Transform.

Calling Sequence

Single precision complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CIFFT (N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZIFFT (N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
Y = FFT-1(X)
```

Description

This routine performs a one-dimensional inverse **FFT** with a maximum vector length of 16384; *N* must be a power of two. The vectors **X** and **Y** may not overlap because the transform is not done in place.

The **CIFFT** routine uses **VPFAST_** as a scratch vector.

The **ZIFFT** routine dynamically allocates a buffer of (3*N*+1) eight-byte words the first time you call it. This buffer is reused if subsequent calls to **ZFFT** use a value of *N* less than or equal to the first, making this routine very fast. If a value of *N* that is larger than the currently allocated buffer is used, the buffer is freed and a larger one is allocated.

If either *INCX* or *INCY* is not 1, **ZIFFT** will run slower. If both *INCX* and *INCY* are not 1, **ZIFFT** allocates an additional 2*N* eight-byte words.

IFIRST()

IFIRST()

Index of first logical true value. (Integer function returning a value.)

Calling Sequence

```
LOGICAL*4 LX(*)
INTEGER*4 N, INCX, I, IFIRST

I = IFIRST (N, LX, INCX)
```

Fortran Equivalent

```
      IFIRST = 0
      DO 10 I = 1, N
         IF (LX(I))
            IFIRST = I
            GOTO 20
         ENDIF
10    CONTINUE
20    CONTINUE
```

ILAST() ILAST()

Index of last logical true value. (Integer function returning a value.)

Calling Sequence

```
LOGICAL*4 LX(*)
INTEGER*4 N, INCX, I, ILAST

I = ILAST (N, LX, INCX)
```

Fortran Equivalent

```
      ILAST = 0
      DO 10 I = 1, N
         IF (LX(I)) ILAST = I
10    CONTINUE
```

IxMAX() IxMAX()

Index of maximum value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDMAX

I = IDMAX(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISMAX

I = ISMAX(N, X, INCX)
```

Fortran Equivalent

```
      IDMAX = 0
      IF (N .GT. 0) IDMAX = 1
      DO 10 I = 2, N
         IF (X(I) .GT. X(IDMAX)) IDMAX = I
10    CONTINUE
```

IxMIN() IxMIN()

Index of minimum value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDMIN

I = IDMIN(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISMIN

I = ISMIN(N, X, INCX)
```

Fortran Equivalent

```
      IDMIN = 0
      IF (N .GT. 0) IDMIN = 1
      DO 10 I = 2, N
         IF (X(I) .LT. X(IDMIN)) IDMIN = I
10    CONTINUE
```

LAND() LAND()

Vector logical AND with vector.

Calling Sequence

```
LOGICAL*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL LAND (N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) .AND. Y(I)
10    CONTINUE
```

LANY() LANY()

Logical true if any elements in *X* are true. (Logical function returning a value.)

Calling Sequence

```
LOGICAL*4 X(*), L, LANY
INTEGER*4 N, INCX

L = LANY (N, X, INCX)
```

Fortran Equivalent

```
      LANY = .FALSE.
      DO 10 I = 1, N
         IF (X(I)) LANY = .TRUE.
10    CONTINUE
```

*x*LBIDI() *x*LBIDI()

Performs the forward substitution to solve the lower bidiagonal in a matrix.

Calling Sequence

Double precision:

```
DOUBLE PRECISION L(*), B(*), X(*)
INTEGER*4 N, INCL, INCB, INCX

CALL DLBIDI(N, L, INCL, B, INCB, X, INCX)
```

Single precision:

```
REAL L(*), B(*), X(*)
INTEGER*4 N, INCL, INCB, INCX

CALL SLBIDI(N, L, INCL, B, INCB, X, INCX)
```

Fortran Equivalent

```
      DO 10 I = 2, N
         X(I) = B(I) - L(I) * X(I-1)
10    CONTINUE
```

Description

*x*LBIDI does the forward substitution to solve AX=B where A is the bidiagonal matrix:

```
[  1   0   0   0       ... 0 ]
[ 12   1   0   0       ... 0 ]
[  0  13   1   0       ... 0 ]
[ ...    ...           ...   ]
[ ...    ...           ...   ]
[ ...    ...     lN-1  1   0 ]
[ ...    ...         lN  1   ]
```

L is the lower diagonal (elements 2 through *N* are used)

B is an input (to solve for) (elements 1 through *N* are used)

X is the output (partial answer) (elements 1 through *N* are used)

*x*LBIDI() *(cont.)*

*x*LBIDI() *(cont.)*

If the matrix is a tridiagonal, you can use *x*TRFAC() to perform an LU factorization of **A**, and then use *x*LBIDI() to solve the lower bidiagonal and *x*UBIDI() to solve the upper bidiagonal. See *x*TRFAC and *x*UBIDI for more information.

LNOT() LNOT()

Vector logical negation.

Calling Sequence

```
LOGICAL*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL LNOT(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = .NOT. X(I)
10    CONTINUE
```

LOR() LOR()

Vector logical OR with vector.

Calling Sequence

```
LOGICAL*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL LOR(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) .OR. Y(I)
10    CONTINUE
```

LSAND() LSAND()

Scalar logical AND with vector.

Calling Sequence

```
LOGICAL*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL LSAND (N, ALPHA, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA .AND. X(I)
10    CONTINUE
```

LSOR() LSOR()

Scalar logical OR with vector.

Calling Sequence

```
LOGICAL*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL LSOR(N, ALPHA, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA .OR. X(I)
10    CONTINUE
```

*x*MASK() *x*MASK()

Conditional assignment.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DMASK(N, W, INCW, X, INCX, LY, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SMASK(N, W, INCW, X, INCX, LY, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         IF (LY(I)) THEN
            Z(I) = W(I)
         ELSE
            Z(I) = X(I)
         ENDIF
10    CONTINUE
```

*x*NE() *x*NE()

Vector element inequality.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DNE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SNE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL INE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LZ(I) = X(I) .NE. Y(I)
10    CONTINUE
```

*x*NEG() *x*NEG()

Change sign.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX

CALL DNEG(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX

CALL SNEG(N, X, INCX)
```

Integer:

```
INTEGER*4 X(*)
INTEGER*4 N, INCX

CALL INEG(N, X, INCX)
```

Complex:

```
COMPLEX X(*)
INTEGER*4 N, INCX

CALL CNEG(N, X, INCX)
```

Double precision complex:

```
COMPLEX*16 X(*)
INTEGER*4 N, INCX

CALL ZNEG(N, X, INCX)
```

*x*NEG() *(cont.)*

*x*NEG() *(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         X(I) = -X(I)
10    CONTINUE
```

*x*NRM2() *x*NRM2()

Euclidean vector norm. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), DNRM2
INTEGER*4 N, INCX

D = DNRM2(N, X, INCX)
```

Single precision:

```
REAL S, X(*), SNRM2
INTEGER*4 N, INCX

S = SNRM2(N, X, INCX)
```

Fortran Equivalent

```
      DNRM2 = 0.0
      DO 10 I = 1, N
         DNRM2 = DNRM2 + X(I)**2
10    CONTINUE
      DNRM2 = SQRT(DNRM2)
```

Description

$$\underline{D}NRM2 = \sqrt{X(1)^2 + X(2)^2 + \ldots + X(N)^2}$$

Note that the summation of the squares operation is performed with scaling to protect from overflow or underflow of intermediate results.

*x*RAMP() *x*RAMP()

Ramp function.

Calling Sequence

Double precision:

```
DOUBLE PRECISION ALPHA, BETA, X(*)
INTEGER*4 N, INCX

CALL DRAMP(N, ALPHA, BETA, X, INCX)
```

Single precision:

```
REAL ALPHA, BETA, X(*)
INTEGER*4 N, INCX

CALL SRAMP(N, ALPHA, BETA, X, INCX)
```

Integer:

```
INTEGER*4 ALPHA, BETA, X(*)
INTEGER*4 N, INCX

CALL IRAMP(N, ALPHA, BETA, X, INCX)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         X(I) = ALPHA + (I-1)*BETA
10    CONTINUE
```

*x*RANDOM() *x*RANDOM()

Pseudo-random number generation. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, DRANDOM
D = DRANDOM()
```

Single precision:

```
REAL S, SRANDOM
S = SRANDOM()
```

Description

The scalar and vector versions of the random number routines use the same seed value stored in:

```
integer seed
common /random_seed/seed
```

Since the same seed is used by all the random routines, a call to the vector routine and *N* calls to the scalar routine would produce the same sequence of numbers (i.e., it is possible to vectorize random number calls and expect the same sequence of random numbers).

The seed's initial value is 1 on each node. The user probably will want to set the seed to a different value on each node. Legal values are in the range from `1` through $2^{31}-1$.

The algorithm used is described in the following article:

Park, Stephen K., and Miller, Keith W.
Random Number Generators: Good Ones are Hard to Find.
Communications of the ACM, Vol. 31, Number 10, (Oct 1988), pg. 1192-1207

*x*ROT() *x*ROT()

Apply a plane rotation.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), C, S
INTEGER*4 N, INCX, INCY

CALL DROT(N, X, INCX, Y, INCY, C, S)
```

Single precision:

```
REAL X(*), Y(*), C, S
INTEGER*4 N, INCX, INCY

CALL SROT(N, X, INCX, Y, INCY, C, S)
```

Fortran Equivalent

```
      DO 10 I = 1, N
      T = X(I)
         X(I) = T * C + Y(I) * S
         Y(I) = -T * S + Y(I) * C
10    CONTINUE
```

Description

Typically, this routine is used to apply a plane rotation to a matrix after using the appropriate *x*ROTG() routine to construct a Givens plane rotation. These subroutines compute the new **X** and **Y** as a function of the sin and cos of the angle through which the matrix is rotated (*C* represents the cosine and *S* represents the sine, the values for which can be obtained with the *x*ROTG() routines):

$$X(I)_{new} = X(I)_{old} * C + Y(I)_{old} * S$$

$$Y(I)_{new} = -X(I)_{old} * S + Y(I)_{old} * C$$

*x*ROTG() *x*ROTG()

Construct a Givens plane rotation. It is intended to be used with *x*ROT().

Calling Sequence

Double precision:

```
DOUBLE PRECISION A, B, C, S

CALL DROTG(A, B, C, S)
```

Single precision:

```
REAL A, B, C, S

CALL SROTG(A, B, C, S)
```

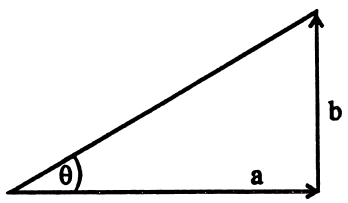
Discussion

Given A and B as the *X* and *Y* values, this routine calculates the sine (*s*) and cosine (*c*) of the angle θ through which the vectors are to be rotated. These values can then be used in the ***x*ROT()** routines to apply the plane rotation.

The routines return *r* overwriting *a*, and *z* overwriting *b*, as well as returning *c* and *s* (when you call the routines, *c* and *s* are uninitialized), where:

$r = \sigma(a^2 + b^2)^{1/2}$

where $\sigma = (\text{sgn}(a) \text{ if } |a| > |b|) \text{ or } (\text{sgn}(b) \text{ if } |a| \geq |b|)$



z is equal to *s*, 1/c, or 1, depending on the following:

s	if $\|a\| > \|b\|$
1/c	if $\|a\| \geq \|b\|$ and $c \neq 0$
1	if $c = 0$

*x*ROTG() *(cont.)*

*x*ROTG() *(cont.)*

If you wish to reconstruct c and s from z, you can do it as follows:

if $z = 0$	set $c = 0$ and $s = 1$
if $\|z\| < 1$	set $c = (1 - z^2)^{1/2}$ and $s = z$
if $\|z\| > 1$	set $c = 1/z$ and $s = (1 - c^2)^{1/2}$

*x*SADD() *x*SADD()

Scalar plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL DSADD(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL SSADD(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ISADD(N, ALPHA, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL CSADD(N, ALPHA, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ZSADD(N, ALPHA, X, INCX, Y, INCY)
```

*x*SADD() *(cont.)*

*x*SADD() *(cont.)*

Fortran Equivalent

```
      DO 10 I = 1,N
         Y(I) = ALPHA + X(I)
10    CONTINUE
```

*x*SCAL() *x*SCAL()

Scalar times a vector to itself.

Calling Sequence

Double precision:

```
DOUBLE PRECISION ALPHA, X(*)
INTEGER*4 N, INCX

CALL DSCAL(N, ALPHA, X, INCX)
```

Single precision:

```
REAL ALPHA, X(*)
INTEGER*4 N, INCX

CALL SSCAL(N, ALPHA, X, INCX)
```

Complex:

```
COMPLEX ALPHA, X(*)
INTEGER*4 N, INCX

CALL CSCAL(N, ALPHA, X, INCX)
```

Double precision complex:

```
COMPLEX*16 ALPHA, X(*)
INTEGER*4 N, INCX

CALL ZSCAL(N, ALPHA, X, INCX)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         X(I) = ALPHA * X(I)
10    CONTINUE
```

SCASUM() SCASUM()

Sum of the absolute values of the real and imaginary parts of a complex vector. (Function returning a value.)

Calling Sequence

Single precision:

```
REAL S, SCASUM
COMPLEX X(*)
INTEGER*4 N, INCX

S = SCASUM(N, X, INCX)
```

Fortran Equivalent

```
      SCASUM = 0.0
      DO 10 I = 1, N
         SCASUM = SCASUM + ABS(REAL(X(I)))+ABS(IMAG(X(I)))
10    CONTINUE
```

Description

$$\text{SCASUM} = \sum_{I=1}^{N} |\text{REAL}(X(I))| + |\text{IMAG}(X(I))|$$

*x*SCATR() *x*SCATR()

Vector scatter.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)

CALL DSCATR(N, X, INCX, IY, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)

CALL SSCATR(N, X, INCX, IY, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)

CALL ISCATR(N, X, INCX, IY, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(IY(I)) = X(I)
10    CONTINUE
```

Note that *IY* must contain values between 1 and *N*. If the values of *IY* are not distinct, the result is undefined.

*x*SDIV() *x*SDIV()

Scalar divided by vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL DSDIV(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL SSDIV(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ISDIV(N, ALPHA, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL CSDIV(N, ALPHA, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ZSDIV(N, ALPHA, X, INCX, Y, INCY)
```

*x*SDIV() *(cont.)*

*x*SDIV() *(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA / X(I)
10    CONTINUE
```

*x*SEQ() *x*SEQ()

Vector equal to scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL DSEQ(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL SSEQ(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL ISEQ(N, ALPHA, X, INCX, LY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .EQ. X(I)
10    CONTINUE
```

*x*SGE() *x*SGE()

Scalar greater than or equal to vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL DSGE(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL SSGE(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL ISGE(N, ALPHA, X, INCX, LY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .GE. X(I)
10    CONTINUE
```

*x*SGT() *x*SGT()

Scalar greater than vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL DSGT(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL SSGT(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL ISGT(N, ALPHA, X, INCX, LY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .GT. X(I)
10    CONTINUE
```

*x*SLE() *x*SLE()

Scalar less than or equal to vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL DSLE(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL SSLE(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL ISLE(N, ALPHA, X, INCX, LY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .LE. X(I)
10    CONTINUE
```

*x*SLT() *x*SLT()

Scalar less than vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL DSLT(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL SSLT(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL ISLT(N, ALPHA, X, INCX, LY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .LT. X(I)
10    CONTINUE
```

*x*SMUL() *x*SMUL()

Scalar times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL DSMUL(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL SSMUL(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ISMUL(N, ALPHA, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL CSMUL(N, ALPHA, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ZSMUL(N, ALPHA, X, INCX, Y, INCY)
```

*x*SMUL() *(cont.)*

*x*SMUL() *(cont.)*

Fortran Equivalent

```
      DO 110 I = 1, N
         Y(I) = ALPHA * X(I)
110   CONTINUE
```

*x*SNE() *x*SNE()

Vector not equal to scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL DSNE(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL SSNE(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY

CALL ISNE(N, ALPHA, X, INCX, LY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .NE. X(I)
10    CONTINUE
```

*x*SOLR() *x*SOLR()

Second-order recurrence routine.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DSOLR(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SSOLR(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I+2) = W(I) + Z(I+1) * X(I) + Z(I) * Y(I)
10    CONTINUE
```

Note that the routine accesses two more elements in **Z** than in **W** or **X**. Be sure to make the **Z** array large enough (including the increment size) to avoid overwriting other memory.

*x*SSUB() *x*SSUB()

Scalar vector subtraction.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL DSSUB(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL SSSUB(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ISSUB(N, ALPHA, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL CSSUB(N, ALPHA, X, INCX, Y, INCY)
```

Double Precision Complex:

```
COMPLEX*16 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY

CALL ZSSUB(N, ALPHA, X, INCX, Y, INCY)
```

*x*SSUB() *(cont.)*

*x*SSUB() *(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA - X(I)
10    CONTINUE
```

*x*SUM() *x*SUM()

Vector sum. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), DSUM
INTEGER*4 N, INCX

D = DSUM(N, X, INCX)
```

Single precision:

```
REAL S, X(*), SSUM
INTEGER*4 N, INCX

S = SSUM(N, X, INCX)
```

Complex:

```
COMPLEX C, X(*), DSUM
INTEGER*4 N, INCX
C = CSUM(N, X, INCX)
```

Double precision complex:

```
COMPLEX*16 Z, X(*), ZSUM
INTEGER*4 N, INCX

Z = ZSUM(N, X, INCX)
```

Fortran Equivalent

```
      DSUM = 0.0
      DO 10 I = 1, N
         DSUM = DSUM + X(I)
10    CONTINUE
```

*x*SUM() *(cont.)*

*x*SUM() *(cont.)*

Description

$$\underline{D}SUM = \sum_{I=1}^{N} X(I)$$

*x*SVMVT()

*x*SVMVT()

Scalar minus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z
(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
CALL DSVMVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVMVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = (ALPHA - X(I)) * Y(I)
10    CONTINUE
```

*x*SVPVT() *x*SVPVT()

Scalar plus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVPVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVPVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = (ALPHA + X(I)) * Y(I)
10    CONTINUE
```

*x*SVTSP()

*x*SVTSP()

Scalar times vector quantity plus scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA, BETA
INTEGER*4 N, INCX, INCY

CALL DSVTSP(N, ALPHA, BETA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA, BETA
INTEGER*4 N, INCX, INCY

CALL SSVTSP(N, ALPHA, BETA, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA * X(I) + BETA
10    CONTINUE
```

*x*SVTVM() *x*SVTVM()

Scalar times vector quantity minus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVTVM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVTVM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ALPHA * X(I) - Y(I)
10    CONTINUE
```

*x*SVTVP() *x*SVTVP()

Scalar times vector quantity plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVTVP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVTVP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ALPHA * X(I) + Y(I)
10    CONTINUE
```

*x*SVVMT() *x*SVVMT()

Scalar times the quantity of vector minus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVVMT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVVMT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ALPHA * (X(I) - Y(I))
10    CONTINUE
```

*x*SVVPT() *x*SVVPT()

Scalar times the quantity of vector plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVVPT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVVPT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ALPHA * (X(I) + Y(I))
10    CONTINUE
```

*x*SVVTM() *x*SVVTM()

Scalar minus the quantity of vector times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVVTM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVVTM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ALPHA - X(I) * Y(I)
10    CONTINUE
```

*x*SVVTP() *x*SVVTP()

Scalar plus the quantity of vector times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL DSVVTP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ

CALL SSVVTP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ALPHA + X(I) * Y(I)
10    CONTINUE
```

*x*SWAP() *x*SWAP()

Swap vectors.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DSWAP(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SSWAP(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ISWAP(N, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CSWAP(N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZSWAP(N, X, INCX, Y, INCY)
```

*x*SWAP() *(cont.)*

*x*SWAP() *(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         T = Y(I)
         Y(I) = X(I)
         X(I) = T
10    CONTINUE
```

*x*TRFAC() *x*TRFAC()

Performs an LU factorization of the tridiagonal of a matrix, and is intended to be used with *x*UBIDI() and *x*LBIDI().

Calling Sequence

Double precision:

```
DOUBLE PRECISION L(*), D(*), U(*)
INTEGER*4 N, INCL, INCD, INCU

CALL DTRFAC(N, L, INCL, D, INCD, U, INCU)
```

Single precision:

```
REAL D(*), U(*), X(*)
INTEGER*4 N, INCD, INCU, INCX

CALL STRFAC(N, L, INCL, D, INCD, U, INCU)
```

Fortran Equivalent

```
      DO 10 I = 2, N
            L(I) = L(I) * D(I-1)
            D(U) = 1.0/(D(I) - L(I) * U(I-1))
10    CONTINUE
```

Discussion

*x*TRFAC does an LU factorization of A, where A is the tridiagonal matrix:

```
[ D1   U1   0   0            ... 0 ]
[ l2   d2  u2   0            ... 0 ]
[  0   l3  d3  u3            ... 0 ]
[ ...  ...                   ...   ]
[ ...  ...                   ...   ]
[ ...  ...          lN-1 DN-1 UN-1 ]
[ ...  ...                 LN   DN ]
```

*x*TRFAC() *(cont.)*

*x*TRFAC() *(cont.)*

The matrix is stored in the following space-efficient way:

L is the lower diagonal	(elements 2 through *N* are used)
D is the diagonal	(elements 1 through *N* are used)
U is the upper diagonal	(elements 1 through *N*-1 are used)

*x*TRFAC() factors the tridiagonal to produce the lower and upper bidiagonals, which you can then solve with *x*LBIDI() and *x*UBIDI(), respectively. In addition, *x*TRFAC() inverts the diagonal, so it is ready for *x*UBIDI() to solve.

*x*UBIDI() *x*UBIDI()

Performs the backward substitution to solve the tridiagonal in a matrix, and is intended to be used with *x*TRFAC() and *x*LBIDI().

Calling Sequence

Double precision:

```
DOUBLE PRECISION D(*), U(*), X(*)
INTEGER*4 N, INCD, INCU, INCX

CALL DUBIDI(N, D, INCD, U, INCU, X, INCX)
```

Single precision:

```
REAL D(*), U(*), X(*)
INTEGER*4 N, INCD, INCU, INCX

CALL SUBIDI(N, D, INCD, U, INCU, X, INCX)
```

Fortran Equivalent

```
      DO 10 I = 2, N
         X(I) = X(I) - U(I) * X(I+1)) * D(I)
10    CONTINUE
```

Discussion

*x*UBIDI does backward substitution to solve Ax=b where A is the bidiagonal matrix:

```
[  D1   U1    0    0        ... 0 ]
[  0    D2   U2    0        ... 0 ]
[  0     0   D3   U3        ... 0 ]
[ ...   ...                 ...   ]
[ ...   ...                 ...   ]
[ ...   ...           0 DN-1 UN-1 ]
[ ...   ...              LN   DN  ]
```

*x*UBIDI()*(cont.)*

*x*UBIDI()*(cont.)*

D is the diagonal	(elements 1 through *N* are used)
U is the upper diagonal	(elements 1 through *N*-1 are used)
X is an input (the output of *x*LBIDI)	(elements 1 through *N* are used) and an output (x that satisfies **Ax=b**)

The *x*UBIDI() routine does a forward substitution to solve the upper bidiagonal.The *x*UBIDI() routine assumes that the diagonal is inverted, done automatically if you have a tridiagonal matrix and use *x*TRFAC() to factor it into upper and lower bidiagonals. If your original matrix has only the upper bidiagonal, invert the diagonal before you use *x*UBIDI().

*x*VABS() *x*VABS()

Element-wise absolute value.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVABS (N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVABS(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL IVABS(N, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*)
REAL SY(*)
INTEGER*4 N, INCX, INCY

CALL CVABS(N, X, INCX, SY, INCY)
```

Double Precision Complex:

```
COMPLEX*16 X(*)
DOUBLE PRECISION DY(*)
INTEGER*4 N, INCX, INCY

CALL ZVABS(N, X, INCX, DY, INCY)
```

*x*VABS() *(cont.)*

*x*VABS() *(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ABS (X(I))
10    CONTINUE
```

Description

$Y(I) = |X(I)|$

*x*VADD() *x*VADD()

Vector addition.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

Complex:

```
COMPLEX X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL CVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL ZVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

*x*VADD()*(cont.)*

*x*VADD()*(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) + Y(I)
10    CONTINUE
```

*x*VAMAX() *x*VAMAX()

Vector element-wise maximum absolute value.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVAMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVAMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = MAX(ABS(X(I)),ABS(Y(I)))
10    CONTINUE
```

Description

$Z(I) = MAX(|X(I)|, |Y(I)|)$

*x*VAMIN() *x*VAMIN()

Vector element-wise minimum absolute value.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVAMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVAMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = MIN(ABS(X(I)),ABS(Y(I)))
10    CONTINUE
```

Description

Z(I) = MIN (| X(I) |, | Y(I) |)

*x*VATAN() *x*VATAN()

Element-wise inverse-tangent.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVATAN(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVATAN(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = ATAN (X(I))
10    CONTINUE
```

Description

Y(I) = arctangent (in radians) of X(I)

*x*VATN2() *x*VATN2()

Element-wise inverse-tangent.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVATN2(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVATN2(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = ATAN2(X(I),Y(I))
10    CONTINUE
```

Description

Z(I) = arctangent (in radians) of X(I) / Y(I)

*x*VCMPLX() *x*VCMPLX()

Construction of a complex vector. The **CVCMPLX()** routine builds a complex vector from two real vectors; the **ZVCMPLX()** routine builds a double precision complex vector from two double precision vectors.

Calling Sequence

Complex:

```
COMPLEX Z(*)
REAL SX(*), SY(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL CVCMPLX(N, SX, INCX, SY, INCY, Z, INCZ)
```

Double precision complex:

```
COMPLEX*16 Z(*)
DOUBLE PRECISION DX(*), DY(*)
INTEGER*4 N, INCX, INCZ

CALL ZVCMPLX(N, DX, INCX, DY, INCY, Z, INCZ)
```

Fortran Equivalent

Real to complex:

```
        DO 10 I = 1, N
           Z(I) = CMPLX(SX(I), SY(I))
10      CONTINUE
```

Double precision to double precision complex:

```
        DO 10 I = 1, N
           Z(I) = CMPLX(DX(I), DY(I))
10      CONTINUE
```

Description

Construction of a complex vector from two vectors, the first *X* being the real part, and the second *Y* being the imaginary part.

*x*VCONJG()

*x*VCONJG()

Calculate the conjugate of a complex vector.

Calling Sequence

Complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CVCONJG(N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZVCONJG(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = CONJG(X(I))
10    CONTINUE
```

*x*VCOS() *x*VCOS()

Element-wise cosine.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVCOS(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVCOS(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = COS(X(I))
10    CONTINUE
```

Description

Y(I) = cosine of X(I) in radians

VDBLE()

VDBLE()

Single to double precision.

Calling Sequence

```
DOUBLE PRECISION DY(*)
INTEGER*4 N, INCX, INCY
REAL SX(*)

CALL VDBLE(N, SX, INCX, DY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         DY(I) = DBLE(SX(I))
10    CONTINUE
```

Description

DY(I) = conversion of SX(I) to double precision

*x*VDIV() *x*VDIV()

Element-wise vector division.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

Complex:

```
COMPLEX X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL CVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL ZVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

*x*VDIV()*(cont.)*

*x*VDIV()*(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) / Y(I)
10    CONTINUE
```

*x*VEXP() *x*VEXP()

Element-wise exponential.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVEXP(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVEXP(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = EXP(X(I))
10    CONTINUE
```

Description

$Y(I) = e^{X(I)}$

*x*VFIX() *x*VFIX()

Truncate elements to integer values.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, IY(*), INCX, INCY

CALL DVFIX(N, X, INCX, IY, INCY)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, IY(*), INCX, INCY

CALL SVFIX(N, X, INCX, IY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         IY(I) = INT(X(I))
10    CONTINUE
```

Description

IY(I) = value resulting from truncation of the fractional part of X(I)

*x*VFLOA() *x*VFLOA()

Convert integer to floating point.

Calling Sequence

Double precision:

```
DOUBLE PRECISION Y(*)
INTEGER*4 N, INCX, INCY, IX(*)

CALL DVFLOA(N, IX, INCX, Y, INCY)
```

Single precision:

```
REAL Y(*)
INTEGER*4 N, INCX, INCY, IX(*)

CALL SVFLOA(N, IX, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = IX(I)
10    CONTINUE
```

Description

Y(I) = conversion of IX(I) to floating point

*x*VIMAG() *x*VIMAG()

Extract the imaginary part of a complex vector.

Calling Sequence

Complex:

```
COMPLEX X(*)
REAL SY(*)
INTEGER N, INCX, INCY

CALL CVIMAG(N, X, INCX, SY, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*)
DOUBLE PRECISION DY(*)
INTEGER N, INCX, INCY

CALL ZVIMAG(N, X, INCX, DY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = IMAG(X(I))
10    CONTINUE
```

*x*VLG10() *x*VLG10()

Element-wise base 10 logarithm.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVLG10(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVLG10(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = LOG10(X(I))
10    CONTINUE
```

Description

$Y(I) = \log_{10}(X(I))$

*x*VLOG() *x*VLOG()

Element-wise natural logarithm.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVLOG(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVLOG(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = LOG(X(I))
10    CONTINUE
```

Description

Y(I) = ln (X(I))

*x*VMAX() *x*VMAX()

Vector element-wise maximum.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = MAX(X(I),Y(I))
10    CONTINUE
```

*x*VMIN() *x*VMIN()

Vector element-wise minimum.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = MIN(X(I),Y(I))
10    CONTINUE
```

*x*VMUL() *x*VMUL()

Element-wise vector multiplication.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

Complex:

```
COMPLEX X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL CVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL ZVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

*x*VMUL()*(cont.)*

*x*VMUL()*(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) * Y(I)
10    CONTINUE
```

*x*VNEG() *x*VNEG()

Negate vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVNEG(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVNEG(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL IVNEG(N, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CVNEG(N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZVNEG(N, X, INCX, Y, INCY)
```

*x*VNEG()*(cont.)*

*x*VNEG()*(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = -X(I)
10    CONTINUE
```

*x*VPOLY() *x*VPOLY()

Vector polynomial evaluation.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), C(*), Y(*)
INTEGER*4 N, M, INCX, INCC, INCY

CALL DVPOLY(N, X, INCX, M, C, INCC, Y, INCY)
```

Single precision :

```
REAL X(*), C(*), Y(*)
INTEGER*4 N, M, INCX, INCC, INCY

CALL SVPOLY(N, X, INCX, M, C, INCC, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = C(M)
10    CONTINUE
      DO 30 J = M-1, 1, -1
         DO 20 I = 1, N
            Y(I) = X(I)*Y(I) + C(J)
20    CONTINUE
30    CONTINUE
```

Description

C is a vector of length *M* containing the coefficients of the polynomial.

Vector polynomial evaluation;

$$Y(I) = C(1) + C(2)X(I) + C(3)X(I)^2 ... + C(M)X(I)^{M-1}$$

Note that if $M \leq 0$, then $Y(I) = 0.0$ for $I = 1, 2, ..., N$

*x*VPOW() *x*VPOW()

Element-wise power function.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVPOW(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVPOW(N, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) ** Y(I)
10    CONTINUE
```

Description

Z(I) = X(I)Y(I)

Note that exponentiation is performed using logarithms. The value of *X(I)* cannot be a negative number since logarithms of negative values are undefined. Therefore, *X(I)* must be greater than or equal to zero. Also, if *X(I)* equals zero, then *Y(I)* must be greater than zero to avoid a divide by zero.

*x*VRANDOM() *x*VRANDOM()

Pseudo-random vector generator.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX

CALL DVRANDOM(N, X, INCX)
```

Single precision :

```
REAL X(*)
INTEGER*4 N, INCX

CALL SVRANDOM(N, X, INCX)
```

Description

The scalar and vector versions of the random number routines use the same seed value stored in:

```
integer seed
common /random_seed/seed
```

Since the same seed is used by all the random routines, a call to the vector routine and *N* calls to the scalar routine would produce the same sequence of numbers. Therefore, it is possible to vectorize random number calls and expect the same sequence of random numbers.

The seed's initial value is `1` on each node. The user probably will want to set the seed to a different value on each node. Legal values are in the range from `1` through $2^{31}-1$.

The algorithm used is described in the following article:

Park, Stephen K., and Miller, Keith W.
Random Number Generators: Good Ones are Hard to Find.
Communications of the ACM, Vol. 31, Number 10, (Oct 1988), pg. 1192-1207

*x*VREAL() *x*VREAL()

Extract the real part of a complex vector.

Calling Sequence

Complex:

```
COMPLEX X(*)
REAL SY(*)
INTEGER N,INCX,INCY

CALL CVREAL(N, X, INCX, SY, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*)
DOUBLE PRECISION DY(*)
INTEGER N,INCX,INCY

CALL ZVREAL(N, X, INCX, DY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = REAL(X(I))
10    CONTINUE
```

*x*VRECP() *x*VRECP()

Vector reciprocal.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVRECP(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVRECP(N, X, INCX, Y, INCY)
```

Complex:

```
COMPLEX X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL CVRECP(N, X, INCX, Y, INCY)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL ZVRECP(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = 1.0 / X(I)
10    CONTINUE
```

*x*VSIN() *x*VSIN()

Element-wise sine.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVSIN(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVSIN(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = SIN(X(I))
10    CONTINUE
```

Description

Y(I) = sine of X(I) in radians

VSNGL() VSNGL()

Double to single precision.

Calling Sequence

```
DOUBLE PRECISION DX(*)
INTEGER*4 N, INCX, INCY
REAL SY(*)

CALL VSNGL(N, DX, INCX, SY, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         SY(I) = SNGL(DX(I))
10    CONTINUE
```

Description

SY(I) = conversion of DX(I) to type **REAL**.

*x*VSQRT() *x*VSQRT()

Element-wise square root.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL DVSQRT(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL SVSQRT(N, X, INCX, Y, INCY)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Y(I) = SQRT(X(I))
10    CONTINUE
```

Description

$$Y(I) = \sqrt{X(I3z)}$$

*x*VSUB() *x*VSUB()

Vector subtraction.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL DVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL SVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL IVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

Complex:

```
COMPLEX X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL CVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

Double precision complex:

```
COMPLEX*16 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ

CALL ZVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

*x*VSUB()*(cont.)*

*x*VSUB()*(cont.)*

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) - Y(I)
10    CONTINUE
```

*x*VVMVT() *x*VVMVT()

Vector minus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DVVMVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SVVMVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = (W(I) - X(I)) * Y(I)
10    CONTINUE
```

*x*VVPVT()

*x*VVPVT()

Vector plus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DVVPVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*),X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SVVPVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = (W(I) + X(I)) * Y(I)
10    CONTINUE
```

*x*VVTVM() *x*VVTVM()

Vector times vector quantity minus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DVVTVM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SVVTVM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = W(I) * X(I) - Y(I)
10    CONTINUE
```

*x*VVTVP()

*x*VVTVP()

Vector times vector quantity plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DVVTVP(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SVVTVP(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = W(I) * X(I) + Y(I)
10    CONTINUE
```

*x*VVVTM() *x*VVVTM()

Vector minus the quantity of vector times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL DVVVTM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ

CALL SVVVTM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Fortran Equivalent

```
      DO 10 I = 1, N
         Z(I) = W(I) - X(I) * Y(I)
10    CONTINUE
```

C VECLIB ROUTINE SUMMARY A

This appendix contains a set of tables that summarize the C VecLib routines according to function. There are eight tables that categorize the routines as follows:

- Mathematical Primitives
- Other Mathematical Functions
- Triad Operations
- Relational Primitive Operations
- Logical Primitive Operations
- Reduction Functions
- Conversion Primitives
- Miscellaneous Operations

The tables describe the functions, providing an example of the calling sequence and the C equivalent, where applicable. In the C equvalent, when *i* is used as an index, *i* ranges from 1 to *n*-1. These routines are descibed in detail in the reference pages in Chapter 9. These reference pages list the routines alphabetically.

The routines are listed by their root names. For most of the functions, there are separate routines for different data types. An italic *x* as the first or second letter of the name represents a letter specifying the data type. These letters are **d** for double, **s** for float (single precision), **i** for integer, **l** for logical, **c** for complex, and **z** for double precision complex. For example, **dasum** is the name of the double precision routine for calculating the sum of absolute values and **sasum** is the single precision version; these are listed under ***x*asum**.

Table A-1. Mathematical Primitives

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*copy	d,s,i,l,c,z	Copy vector	`dcopy(n,x,incx,y,incy);` `y[i] = x[i];`
*x*fill	d,s,i,l,c,z	Fill vector	`dfill(n,alpha,x,incx);` `x[i] = alpha;`
*x*neg	d,s,i,c,z	Change sign	`dneg(n,x,incx);` `x[i] = -x[i];`

*x*swap	d,s,i,c,z	Swap vectors	`dswap(n,x,incx,y,incy);`
			`t = y[i];`
			`y[i] = x[i];`
			`x[i] = t;`

*x*sadd	d,s,i,c,z	Scalar plus vector	`dsadd(n,alpha,x,incx,y,incy);` `y[i] = alpha + x[i];`
*x*scal	d,s,c,z	Scalar times a vector to itself	`dscal(n,alpha,x,incx);` `x[i] = alpha * x[i];`
*x*sdiv	d,s,i,c,z	Scalar divided by vector	`dsdiv(n,alpha,x,incx,y,incy);` `y[i] = alpha / x[i];`
*x*smul	d,s,i,c,z	Scalar times vector	`dsmul(n,alpha,x,incx,y,incy);` `y[i] = alpha * x[i];`
*x*ssub	d,s,i,c,z	Scalar vector subtraction	`dssub(n,alpha,x,incx,y,incy);` `y[i] = alpha - x[i];`
*x*vadd	d,s,i,c,z	Vector addition	`dvadd(n,x,incx,y,incy,z,incz);` `z[i] = x[i] + y[i];`
*x*vdiv	d,s,i,c,z	Element-wise vector division	`dvdiv(n,x,incx,y,incy,z,incz);` `z[i] = x[i] / y[i];`
*x*vmul	d,s,i,c,z	Vector element-wise multiplication	`dvmul(n,x,incx,y,incy,z,incz);` `z[i] = x[i] * y[i];`
*x*vneg	d,s,i,c,z	Negate vector	`dvneg(n,x,incx,y,incy);` `y[i] = -x[i];`
*x*vrecp	d,s	Vector reciprocal	`dvrecp(n,x,incx,y,incy);` `y[i] = 1.0 / x[i];`
*x*vsub	d,s,i,c,z	Vector subtraction	`dvsub(n,x,incx,y,incy,z,incz);` `z[i] = x[i] - y[i];`

Table A-2. Other Mathematical Functions

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
***x*random**	d,s	Generates pseudo-random number	`d = drandom();`
***x*vabs**	d,s,i,c,z	Element-wise absolute value	`dvabs(n,x,incx,y,incy);` `y[i] = CABS(x[i]);`
***x*vamax**	d,s	Vector element-wise maximum absolute value	`dvamax(n,x,incx,y,incy,z,incz);` `z[i] = MAX(ABS(x[i]),ABS(y[i]));`
***x*vamin**	d,s	Vector element-wise minimum absolute value	`dvamin(n,x,incx,y,incy,z,incz);` `z[i] = MIN(ABS(x[i]),ABS(y[i]));`
***x*vatan**	d,s	Element-wise inverse-tangent	`dvatan(n,x,incx,y,incy);` `y[i] = atan(x[i]);`
***x*vatn2**	d,s	Element-wise inverse-tangent of quotient	`dvatn2(n,x,incx,y,incy,z,incz);` `z[i] = atan2(x[i],y[i]);`
***x*vcos**	d,s	Element-wise cosine	`dvcos(n,x,incx,y,incy);` `y[i] = cos(x[i]);`
***x*vexp**	d,s	Element-wise exponential	`dvexp(n,x,incx,y,incy);` `y[i] = exp(x[i]);`
***x*vlg10**	d,s	Element-wise base 10 logarithm.	`dvlg10(n,x,incx,y,incy);` `y[i] = log10(x[i]);`
***x*vlog**	d,s	Element-wise natural logarithm	`dvlog(n,x,incx,y,incy);` `y[i] = log(x[i]);`
***x*vmax**	d,s	Vector element-wise maximum	`dvmax(n,x,incx,y,incy,z,incz);` `z[i] = MAX(x[i],y[i]);`
***x*vmin**	d,s	Vector element-wise minimum	`dvmin(n,x,incx,y,incy,z,incz);` `z[i] = MIN(x[i],y[i]);`
***x*vpow**	d,s	Element-wise power function	`dvpow(n,x,incx,y,incy,z,incz);` `z[i] = pow(x[i], y[i]);`
***x*vrandom**	d,s c,z	Pseudo-random vector generation	`dvrandom(n,x,incx);`
***x*vsin**	s	Element-wise sine	`dvsin(n,x,incx,y,incy);` `y[i] = sin(x[i]));`
***x*vsqrt**	d,s	Element-wise square root	`dvsqrt(n,x,incx,y,incy);` `y[i] = sqrt(x[i]);`

Table A-3. Triad Operations

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*axpy	d,s,c,z	(Scalar x vector) + vector	daxpy(n,alpha,x,incx,y,incy); y[i] = alpha*x[i] + y[i];
*x*svmvt	d,s	(Scalar - vector)vector	dsvmvt(n,alpha,x,incx,y,incy,z,incz); z[i] = (alpha - x[i]) * y(i);
*x*svpvt	d,s	(Scalar + vector)vector	dsvpvt(n,alpha,x,incx,y,incy,z,incz); z[i] = (alpha + x[i]) * y[i];
*x*svtsp	d,s	(Scalar x vector) + scalar	dsvtsp(n,alpha,beta,x,incx,y,incy); y[i] = alpha * x[i] + beta;
*x*svtvm	d,s	(Scalar x vector) - vector	dsvtvm(n,alpha,x,incx,y,incy,z,incz); z[i] = alpha * x[i] - y[i];
*x*svtvp	d,s	(Scalar x vector) + vector	dsvtvp(n,alpha,x,incx,y,incy,z,incz); z[i] = alpha * x[i] + y[i];
*x*svvmt	d,s	Scalar x (vector - vector)	dsvvmt(n,alpha,x,incx,y,incy,z,incz); z[i] = alpha * (x[i] - y[i]);
*x*svvpt	d,s	Scalar x (vector + vector)	dsvvpt(n,alpha,x,incx,y,incy,z,incz); z[i] = alpha * (x[i] + y[i]);
*x*svvtm	d,s	Scalar - (vector x vector)	dsvvtm(n,alpha,x,incx,y,incy,z,incz); z[i] = alpha - x[i] * y[i];
*x*svvtp	d,s	Scalar + (vector x vector)	dsvvtp(n,alpha,x,incx,y,incy,z,incz); z[i] = alpha + x[i] * y[i];
*x*vvmvt	d,s	(Vector - vector) x vector	dvvmvt(n,w,incw,x,incx,y,incy,z,incz); z[i] = (w[i] - x[i]) * y[i];
*x*vvpvt	d,s	(Vector + vector x vector	dvvpvt(n,w,incw,x,incx,y,incy,z,incz); z[i] = (w[i] + x[i]) * y[i];
*x*vvtvm	d,s	(Vector x vector) - vector	dvvtvm(n,w,incw,x,incx,y,incy,z,incz); z[i] = w(i) * x[i] - y[i];
*x*vvtvp	d,s	(Vector x vector) + vector	dvvtvp(n,w,incw,x,incx,y,incy,z,incz); z[i] = w[i] * x[i] + y[i];
*x*vvvtm	d,s	Vector - (vector x vector	dvvvtm(n,w,incw,x,incx,y,incy,z,incz); z[i] = w[i] - x[i] * y[i];

Table A-4. Relational Primitive Operations

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*eq	d,s,i	Vector element equality	`deq(n,x,incx,y,incy,lz,incz);` `lz[i] = x[i] == y[i];`
*x*ge	d,s,i	Vector element greater than or equal to	`dge(n,x,incx,y,incy,lz,incz);` `lz[i] = x[i] >= y[i];`
*x*gt	d,s,i	Vector element greater than	`dgt(n,x,incx,y,incy,lz,incz);` `lz[i] = x(i) > y[i];`
*x*ne	d,s,i	Vector element inequality	`dne(n,x,incx,y,incy,lz,incz);` `lz[i] = x[i] != y[i];`
*x*seq	d,s,i	Vector equal to scalar	`dseq(n,alpha,x,incx,ly,incy);` `ly[i] = alpha == x[i];`
*x*sge	d,s,i	Scalar greater than or equal to vector	`dsge(n,alpha,x,incx,ly,incy);` `ly[i] = alpha >= x[i];`
*x*sgt	d,s,i	Scalar greater than vector	`dsgt(n,alpha,x,incx,ly,incy);` `ly[i] = alpha > x[i];`
*x*sle	d,s,i	Scalar less than or equal to vector	`dsle(n,alpha,x,incx,ly,incy);` `ly[i] = alpha <= x[i];`
*x*slt	d,s,i	Scalar less than vector	`dslt(n,alpha,x,incx,ly,incy);` `ly[i] = alpha < x[i];`
*x*sne	d,s,i	Vector not equal to scalar	`dsne(n,alpha,x,incx,ly,incy);` `ly[i] = alpha != x[i];`

Table A-5. Logical Primitive Operations

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
land	1	Vector logical AND with vector	`land(n,x,incx,y,incy,z,incz);` `z[i] = x[i] && y[i];`
lcopy	1	Copy vector	`lcopy(n,x,incx,y,incy);` `y[i] = x[i];`
lfill	1	Fill vector	`lfill(n,alpha,x,incx);` `x[i] = alpha;`
lnot	1	Vector logical negation	`lnot(n,x,incx,y,incy);` `y[i] = !x[i];`
lor	1	Vector logical OR with vector	`lor(n,x,incx,y,incy,z,incz);` `z[i] = x[i] y[i];`
lsand	1	Scalar logical AND with scalar	`lsand(n,alpha,x,incx,y,incy);` `y[i] = alpha || x[i];`
lsor	1	Scalar logical OR with vector	`lsor(n,alpha,x,incx,y,incy);` `y[i] = alpha x[i];`

Table A-6. Reduction Functions (1 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
***x*asum**	d,s	Sum of absolute values	`d = dasum(n,x,incx);` `dasum = dasum + ABS(x[i]);`
***x*dot**	d,s	Dot product of two vectors	`d = ddot(n,x,incx,y,incy);` `ddot = ddot + x[i]*y[i];`

***x*dotc**	c,z	Dot product of two complex vectors, first vector conjg.	`c = cdotc(n,x,incx,y,incy);`
			`cdotc = CMPLX(0.0,0.0);`
			`cdotc = cdotc + (CONJG(x[i])*y[i]);`

***x*dotu**	c,z	Dot product of two complex vectors	`c = cdotu(n,x,incx,y,incy);` `cdotu = cdotu + x[i]*y[i];`

dzasum	d	Sum of absolute value of real and imaginary parts	`d = dzasum(n,x,incx);`
			`dzasum = dzasum + ABS(x[i].r) +`
			`ABS(x[i].i);`

ixamax	d,s	Index of maximum absolute value	`i = idamax(n,x,incx);`
			`if (n>0) idamax = 0;`
			`if (ABS(x[i])>ABS(x[idamax])) idamax=i;`

ixamin	d,s	Index of minimum absolute value	`i = idamin(n,x,incx);`
			`idamin = min (1, max(0,n));`
			`if (ABS(x[i])<ABS(x[idamin])) idamin=i;`

icount	l	Number of logical true values	`i = icount(n, lx, incx);`
			`icount = 0;`
			`if (lx[i]) icount = icount + 1;`

ifirst	i	Index of first logical true value	`i = ifirst(n,lx,incx);` `ifirst = -1; if (lx[i]) {ifirst = i;break;};`
ilast	i	Index of last logical true value	`i = ilast(n,lx,incx);` ilast = -1; if (lx[i]) ilast = i;

ixmax	d,s	Index of maximum value	`i = idmax(n,x,incx);`
			`idmax = -1;if(n>0) idmax = 0;`
			`if (x[i] > x[idmax]) idmax = i;`

ixmin	d,s	Index of minimum value	`i = idmin(n,x,incx);`
			`idmin = -1;if (n>0) idmin=0;`
			`if (x[i] < x[idmin]) idmin = i;`

lany	l	Logical true if any of x are true	`l = lany(n,x,incx);`
			`lany = 0;`
			`if (x[i]) lany = 1;`

***x*nrm2**	d,s	Euclidean vector norm	`d = dnrm2(n,x,incx);`
			`dnrm2 = dnrm2 + x[i]*x[i];`
			`dnrm2 = sqrt(dnrm2);`

Table A-6. Reduction Functions (2 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent

scasum	s	Sum of abs val of real and imag parts of the vector	`s = scasum(n,x,incx);`
			`scasum = scasum + ABS(x[i].r) +`
			`ABS(x[i].i);`

***x*sum**	d,s,c,z	Vector sum	`s = dsum(n,x,incx);` `dsum = dsum + x[i];`

Table A-7. Conversion Primitives

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
***x*vcmplx**	c,z	Real to complex	`cvcmplx(n,sx,incx,sy,incy,z,incz);` `z[i] = CMPLX(sx[i],sy[i]);`
***x*vconjg**	c,z	Conjugate of a complex vector	`cvconjg(n,x,incx,y,incy);` `y[i] = CONJG(x[i]);`
vdble	d	Single to double precision	`vdble(n,sx,incx,dy,incy);` `dy[i] = (double)sx[i];`
***x*vfix**	d,s	Truncate elements to integer values	`dvfix(n,x,incx,iy,incy);` `iy[i] = (int)x[i];`
***x*vfloa**	d,s	Convert integer to floating point	`dvfloa(n,ix,incx,y,incy);` `y[i] = ix[i];`
***x*vimag**	c,z	Extract imaginary part of complex vector	`cvimag(n,x,incx,sy,incy);` `sy[i] = x[i].i;`
***x*vreal**	d,s,c,z	Extract real part of complex vector	`cvreal(n,x,incx,sy,incy);` `sy[i]= x[i].r;`
vsngl	s	Double to single precision	`vsngl(n,dx,incx,sy,incy);` `sy[i] = (float)dx[i];`

Table A-8. Miscellaneous Functions (1 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*clip	d,s	Clip to interval [alpha,beta]	`dclip(n,x,incx,alpha,beta,y,incy);` `y[i] = MIN(MAX(x[i],alpha),beta);`
*x*cndst	d,s	Conditional assignment	`dcndst(n,x,incx,ly,incy,z,incz);` `if (ly[i]) z[i] = x[i];`
*x*fft	c,z	Fast Fourier Transform	`cfft(n,x,incx,y,incy);` `y = fft(x);`
*x*folr	d,s	First order linear recurrence routine	`dfolr(n,x,incx,y,incy,z,incz);` `z[i+1] = y[i] + z[i] * x[i];`
*x*gathr	d,s,i	Vector gather	`dgathr(n,x,incx,iy,incy,z,incz);` `z[i] = x[iy[i]];`

*x*iclip	d,s	Inverted clip	`diclip(n,x,incx,alpha,beta,y,incy);`
			`if (x[i] < (alpha + beta)/2.0)`
			`  y[i] = MIN(x[i],alpha);`
			`else y[i] = MAX(x[i],beta);`

*x*ifft	c,z	Inverse Fast Fourier Transform	`y = cifft(n,x,incx,y,incy);` `y = ifft-1(x);`
*x*lbidi	d,s	Solves lower bidiagonal with fwd. Substitution	`dlbidi(n,1,incl,b,incb,x,incx);` `x[i] = b[i] - l[i] * x[i-1];`
*x*mask	d,s	Conditional assignment	`dmask(n,w,incw,x,incx,ly,incy,z,incz);` `if (ly[i]) z[i] = w[i]; else z[i] = x[i];`
*x*ramp	d,s,i	Ramp function	`dramp(n,alpha,beta,x,incx);` `x[i] = alpha + (i-1)*beta;`

*x*rot	d,s	Apply a plane rotation	`drot(n,x,incx,y,incy,c,s);`
			`x[i] = t * c + y[i] * s;`
			`y[i] = -t * s + y[i] * c;`

*x*rotg	d,s	Construct a Givens plane rotation	`drotg(a, b, c, s);`
*x*scatr	d,s,i	Vector scatter	`dscatr(n,x,incx,iy,incy,z,incz);` `z[iy[i]] = x[i];`
*x*solr	d,s	Second order recurrence routine	`dsolr(n,w,incw,x,incx,y,incy,z,incz);` `z[i+2] = w[i]+z[i+1]*x[i]+z[i] * y[i];`

Table A-8. Miscellaneous Functions (2 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent

*x*trfac	d,s	LU factorization of matrix tridiagonal	`dtrfac(n,l,incl,d,incd,u,incu);`
			`l[i] = l[i] * d[i-1];`
			`d[i] = 1.0/(d[i] - l[i] * u[i-1]);`

*x*ubidi	d,s	Solves upper bidiagonal	`dubidi(n,d,incd,u,incu,x,incx);` `u[i+2]=(u[i]-d[i]) * u[i+1]) *l[i];`
*x*vpoly	d,s	Vector polynomial evaluation	`dvpoly(n,x,incx,m,c,incc,y,incy);` `y[i] = x[i] * y[i] + c[j];`

FORTRAN VECLIB ROUTINE SUMMARY B

This appendix contains a set of tables that summarize the Fortran VecLib routines according to function. There are eight tables that categorize the routines as follows:

- Mathematical Primitives
- Other Mathematical Functions
- Triad Operations
- Relational Primitive Operations
- Logical Primitive Operations
- Reduction Functions
- Conversion Primitives
- Miscellaneous Operations

The tables describe the functions, providing an example of the calling sequence and the Fortran equivalent, where applicable. In the Fortran equvalent, when *i* is used as an index, *i* ranges from 1 to *n*. These routines are descibed in detail in the reference pages in Chapter 8. These reference pages list the routines alphabetically.

The routines are listed by their root names. For most of the functions, there are separate routines for different data types. An italic *x* as the first or second letter of the name represents a letter specifying the data type. These letters are **d** for double, **s** for single, **i** for integer, **l** for logical, **c** for complex, and **z** for double precision complex. For example, **dasum** is the name of the double precision routine for calculating the sum of absolute values and **sasum** is the single precision version; these are listed under ***x*asum**.

Table B-1. Mathematical Primitives

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*copy	d,s,i,l,c,z	Copy vector	`call dcopy(n,x,incx,y,incy)` `y(i) = x(i)`
*x*fill	d,s,i,l,c,z	Fill vector	`call dfill(n,alpha,x,incx)` `x(i) = alpha`
*x*sadd	d,s,i,c,z	Scalar + vector	`call dsadd(n,alpha,x,incx,y,incy)` `y(i) = alpha + x(i)`
*x*scal	d,s,c,z	Scalar times a vector to itself	`call dscal(n,alpha,x,incx)` `x(i) = alpha * x(i)`
*x*ssub	d,s,i,c,z	Scalar vector subtraction	`call dssub(n,alpha,x,incx,y,incy)` `y(i) = alpha - x(i)`

*x*neg	d,s,i,c,z	Change sign	dasum = dasum + abs(x(i))
			`call dneg(n,x,incx)`
			`x(i) = -x(i)`

*x*sdiv	d,s,i,c,z	Scalar divided by vector	`call dsdiv(n,alpha,x,incx,y,incy)` `y(i) = alpha / x(i)`
*x*smul	d,s,i,c,z	Scalar times vector	`call dsmul(n,alpha,x,incx,y,incy)` `y(i) = alpha * x(i)`
*x*swap	d,s,i,c,z	Swap vectors	`call dswap(n,x,incx,y,incy)` `t = y(i) y(i) = x(i) x(i) = t`
*x*vadd	d,s,i,c,z	Vector addition	`call dvadd(n,x,incx,y,incy,z,incz)` `z(i) = x(i) + y(i)`
*x*vdiv	d,s,i,c,z	Element-wise vector division	`call dvdiv(n,x,incx,y,incy,z,incz)` `z(i) = x(i) / y(i)`
*x*vmul	d,s,i,c,z	Vector element-wise multiplication	`call dvmul(n,x,incx,y,incy,z,incz)` `z(i) = x(i) * y(i)`
*x*vneg	d,s,i,c,z	Negate vector	`call dvneg(n,x,incx,y,incy)` y(i) = -x(i)
*x*vrecp	d,s	Vector reciprocal	`call dvrecp(n,x,incx,y,incy)` `y(i) = 1.0 / x(i)`
*x*vsub	d,s,i,c,z	Vector subtraction	`call dvsub(n,x,incx,y,incy,z,incz)` `z(i) = x(i) - y(i)`

Table B-2. Other Mathematical Functions

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
***x*random**	d,s	Generates pseudo-random number	`d = drandom()`
***x*vabs**	d,s,i,c,z	Element-wise absolute value	`call dvabs(n,x,incx,y,incy)` `y(i) = abs(x(i))`
***x*vamax**	d,s	Vector element-wise maximum absolute value	`call dvamax(n,x,incx,y,incy,z,incz)` `z(i) = MAX(abs(x(i)),abs(y(i)))`
***x*vamin**	d,s	Vector element-wise minimum absolute value	`call dvamin(n,x,incx,y,incy,z,incz)` `z(i) = MIN(abs(x(i)),abs(y(i)))`
***x*vatan**	d,s	Element-wise inverse-tangent	`call dvatan(n,x,incx,y,incy)` `y(i) = atan(x(i))`
***x*vatn2**	d,s	Element-wise inverse-tangent of quotient	`call dvatn2(n,x,incx,y,incy,z,incz)` `z(i) = atan2(x(i),y(i))`
***x*vcos**	d,s	Element-wise cosine	`call dvcos(n,x,incx,y,incy)` `y(i) = cos(x(i))`
***x*vexp**	d,s	Element-wise exponential	`call dvexp(n,x,incx,y,incy)` `y(i) = exp(x(i))`
***x*vlg10**	d,s	Element-wise base 10 logarithm.	`call dvlg10(n,x,incx,y,incy)` `y(i) = log10(x(i))`
***x*vlog**	d,s	Element-wise natural logarithm	`call dvlog(n,x,incx,y,incy)` `y(i) = log(x(i))`
***x*vmax**	d,s	Vector element-wise maximum	`call dvmax(n,x,incx,y,incy,z,incz)` `z(i) = MAX(x(i),y(i))`
***x*vmin**	d,s	Vector element-wise minimum	`call dvmin(n,x,incx,y,incy,z,incz)` `z(i) = MIN(x(i),y(i))`
***x*vpow**	d,s	Element-wise power function	`call dvpow(n,x,incx,y,incy,z,incz)` `z(i) = x(i) ** y(i)`
***x*vrandom**	d,s c,z	Pseudo-random vector generation	`call dvrandom(n,x,incx)`
***x*vsin**	s	Element-wise sine	`call dvsin(n,x,incx,y,incy)` `y(i) = sin(x(i))`
***x*vsqrt**	d,s	Element-wise square root	`call dvsqrt(n,x,incx,y,incy)` `y(i) = sqrt(x(i))`

Table B-3. Triad Operations

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*axpy	d,s,c,z	(Scalar x vector) + vector to itself	`call daxpy(n,alpha,x,incx,y,incy)` `y(i) = alpha*x(i) + y(i)`
*x*svmvt	d,s	(Scalar - vector)vector	`call dsvmvt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = (alpha - x(i)) * y(i)`
*x*svpvt	d,s	(Scalar + vector)vector	`call dsvpvt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = (alpha + x(i)) * y(i)`
*x*svtsp	d,s	(Scalar x vector) + scalar	`call dsvtsp(n,alpha,beta,x,incx,y,incy)` `y(i) = alpha * x(i) + beta`
*x*svtvm	d,s	(Scalar x vector) - vector	`call dsvtsm(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * x(i) - y(i)`
*x*svtvp	d,s	(Scalar x vector) + vector	`call dsvtvp(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * x(i) + y(i)`
*x*svvmt	d,s	Scalar x (vector - vector)	`call dsvvmt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * (x(i) - y(i))`
*x*svvpt	d,s	Scalar x (vector + vector)	`call dsvvpt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * (x(i) + y(i))`
*x*svvtm	d,s	Scalar - (vector x vector)	`call dsvvtm(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha - x(i) * y(i)`
*x*svvtp	d,s	Scalar + (vector x vector)	`call dsvvtp(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha + x(i) * y(i)`
*x*vvmvt	d,s	(Vector - vector) x vector	`call dvvmvt(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = (w(i) - x(i)) * y(i)`
*x*vvpvt	d,s	(Vector + vector x vector	`call dvvpvt(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = (w(i) + x(i)) * y(i)`
*x*vvtvm	d,s	(Vector x vector) - vector	`call dvvtvm(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = w(i) * x(i) - y(i)`
*x*vvtvp	d,s	(Vector x vector) + vector	`call dvvtvp(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = w(i) * x(i) + y(i)`
*x*vvvtm	d,s	Vector - (vector x vector	`call dvvvtm(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = w(i) - x(i) * y(i`

Table B-4. Relational Primitive Operations

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*eq	d,s,i	Vector element equality	`call deq(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .eq. y(i)`
*x*ge	d,s,i	Vector element greater than or equal to	`call dge(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .ge. y(i)`
*x*gt	d,s,i	Vector element greater than	`call dgt(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .gt. y(i)`
*x*ne	d,s,i	Vector element inequality	`call dne(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .ne. y(i)`
*x*seq	d,s,i	Vector equal to scalar	`call dseq(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .eq. x(i)`
*x*sge	d,s,i	Scalar greater than or equal to vector	`call dsge(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .ge. x(i)`
*x*sgt	d,s,i	Scalar greater than vector	`call dsgt(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .gt. x(i)`
*x*sle	d,s,i	Scalar less than or equal to vector	`call dsle(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .le. x(i)`
*x*slt	d,s,i	Scalar less than vector	`call dslt(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .lt. x(i)`
*x*sne	d,s,i	Vector not equal to scalar	`call dsne(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .ne. x(i)`

Table B-5. Logical Primitive Operations

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
land	l	Vector logical AND with vector	`call land(n,x,incx,y,incy,z,incz)` `z(i) = x(i) .and. y(i)`
lcopy	l	Copy vector	`call lcopy(n,x,incx,y,incy)` `y(i) = x(i)`
lfill	l	Fill vector	`call lfill(n,alpha,x,incx)` `x(i) = alpha`
lnot	l	Vector logical negation	`call lnot(n,x,incx,y,incy)` `y(i) = .not. x(i)`
lor	l	Vector logical OR with vector	`call lor(n,x,incx,y,incy,z,incz)` `z(i) = x(i) .or. y(i)`
lsand	l	Scalar logical AND with scalar	`call lsand(n,alpha,x,incx,y,incy)` `y(i) = alpha .and. x(i)`
lsor	l	Scalar logical OR with vector	`call lsor(n,alpha,x,incx,y,incy)` `y(i) = alpha .or. x(i)`

Table B-6. Reduction Functions (1 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
***x*asum**	d,s	Sum of absolute values	`d = dasum(n,x,incx)` dasum = dasum + abs(x(i))
***x*dot**	d,s	Dot product of two vectors	`d = ddot (n,x,incx,y,incy)` `ddot = ddot + x(i)*y(i)`

***x*dotc**	c,z	Dot product of two complex vectors, first vector conjg.	`c = cdotc (n,x,incx,y,incy)`
			`cdotc = (0.0,0.0)`
			`cdotc = cdotc + CONJG(x(i))*y(i)`

***x*dotu**	c,z	Dot product of two complex vectors	`c = cdotu (n,x,incx,y,incy)` `cdotu = cdotu + x(i)*y(i)`

dzasum	d	Sum of absolute value of real and imaginary parts	`d = dzasum(n,x,incx)`
			`dzasum = dzasum + abs(real(x(i))) +`
			`abs(imag(x(i)))`

***x*amax**	d,s	Index of maximum absolute value	`i = idamax(n,x,incx)`
			`idamax = MIN (1, max(0,n))`
			`if(abs(x(i)).gt.abs(x(idamax)))idamax=i`

ixamin	d,s	Index of minimum absolute value	`i = idamin(n,x,incx)`
			`idamin = MIN(1, max(0,n))`
			`if(abs(x(i)).lt.abs(x(idamin)))idamin=i`

icount	l	Number of logical true values	`i = icount(n,lx,incx)`
			`icount = 0`
			`if (lx(i)) icount = icount + 1`

ifirst	i	Index of first logical true value	`i = ifirst(n,lx,incx)` `if (lx(i)) ifirst = i`
ilast	i	Index of last logical true value	`i = ilast(n,lx,incx)` if (lx(i) .and. ifirst .eq. 1)ilast = i

ixmax	d,s	Index of maximum value	`i = idmax(n,x,incx)`
			`idmax = min (1, max(0,n))`
			`if (x(i) .gt. x(idmax)) idmax = i`

ixmin	d,s	Index of minimum value	`i = idmin(n,x,incx)`
			`idmin = min (1, max(0,n))`
			`if (x(i) .lt. x(idmin)) idmin = i`

lany	l	Logical true if any of x are true	`l = lany(n,x,incx)`
			`lany = .false.`
			`if (x(i)) lany = .true.`

***x*nrm2**	d,s	Euclidean vector norm	`d = dnrm2(n,x,incx)` `d = sqrt(ddot(n,x,incx,x,incx))`

Table B-6. Reduction Functions (2 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent

scasum	s	Sum of abs val of real and imag parts of the vector	`s = scasum(n,x,incx)`
			`scasum = scasum + abs(real(x(i))) +`
			`abs(imag(x(i)))`

***x*sum**	d,s,c,z	Vector sum	`s = dsum(n,x,incx)` `dsum = dsum + x(i)`

Table B-7. Conversion Primitives

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
***x*vcmplx**	c,z	Real to complex	`call cvcmplx(n,sx,incx,sy,incy,z,incz)` `z(i) = CMPLX(sx(i),sy(i))`
***x*vconjg**	c,z	Conjugate of a complex vector	`call cvconjg(n,x,incx,y,incy)` `y(i) = CONJG(x(i))`
vdble	d	Single to double precision	`call vdble(n,sx,incx,dy,incy)` `dy(i) = dble(sx(i))`
***x*vfix**	d,s	Truncate elements to integer values	`call dvfix(n,x,incx,iy,incy)` `iy(i) = int(x(i))`
***x*vfloa**	d,s	Convert integer to floating point	`call dvfloa(n,ix,incx,y,incy)` `y(i) = ix(i)`
***x*vimag**	c,z	Extract imaginary part of complex vector	`call cvimag(n,x,incx,sy,incy)` `sy(i) = imag(x(i))`
***x*vreal**	d,s,c,z	Extract real part of complex vector	`call cvreal(n,x,incx,sy,incy)` `sy(i) = real(x(i))`
vsngl	s	Double to single precision	`call vsngl(n,dx,incx,sy,incy)` `sy(i) = sngl(dx(i))`

Table B-8. Miscellaneous Functions (1 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent
*x*clip	d,s	Clip to interval [alpha,beta]	`call dclip(n,x,incx,alpha,beta,y,incy)` `y(i) = MAX(MAX(x(i),alpha),beta)`
*x*cndst	d,s	Conditional assignment	`call dcndst(n,x,incx,ly,incy,z,incz)` `if (ly(i)) z(i) = x(i)`
*x*fft	c,z	Fast Fourier Transform	`call cfft(n,x,incx,y,incy)` `y = fft(x)`
*x*folr	d,s	First order linear recurrence routine	`call dfolr(n,x,incx,y,incy,z,incz)` `z(i+1) = y(i)+z(i)*x(i)`
*x*gathr	d,s,i	Vector gather	`call dgathr(n,x,incx,iy,incy,z,incz)` `z(i) = x(iy(i))`

*x*iclip	d,s	Inverted clip	`call diclip(n,x,incx,alpha,beta,y,incy)`
			`if (x(i).lt.(alpha + beta)/2.0) then`
			`  y(i) = MIN (x(i),alpha)`
			`else y(i) = MAX(x(i),beta)`

*x*ifft	c,z	Inverse Fast Fourier Transform	`y = cifft(n,x,incx,y,incy)` `y = fft-1(x)`
*x*lbidi	d,s	Solves lower bidiagonal with fwd. Substitution	`call dlbidi(n,l,incl,b,incb,x,incx)` `x(i) = b(i) - l(i) * x(i - 1)`
*x*mask	d,s	Conditional assignment	`call dmask(n,w,incw,x,incx,ly,incy,z,incz)` `if (ly(i)) z(i) = w(i) else z(i) = x(i)`
*x*ramp	d,s,i	Ramp function	`call dramp(n,alpha,beta,x,incx)` `x(i) = alpha + (i-1)*beta`

*x*rot	d,s	Apply a plane rotation	`call drot(n,x,incx,y,incy,c,s)`
			`x(i) = t * c + y(i) * s`
			`y(i) = -t * s + y(i) * c`

*x*rotg	d,s	Construct a Givens plane rotation	`call drotg(a, b, c, s)`
*x*scatr	d,s,i	Vector scatter	`call dscatr(n,x,incx,iy,incy,z,incz)` `z(iy(i)) = x(i)`
*x*solr	d,s	Second order recurrence routine	`call dsolr(n,w,incw,x,incx,y,incy,z,incz)` `z(i+2) = w(i)+z(i + 1)*x(i)+z(i)*y(i)`

Table B-8. Miscellaneous Functions (2 of 2)

Root Name	Data Type	Description	Calling Sequence/ Fortran Equivalent

*x*trfac	d,s	LU factorization of matrix tri-diagonal	`call dtrfac(n,l,incl,d,incd,u,incu)`
			`l(i) = l(i) * d(i-1)`
			`d(i) = 1.0/(d(i) - l(i) * u(i-1))`

*x*ubidi	d,s	Solves upper bidiagonal	`call dubidi(n,d,incd,u,incu,x,incx)` `u(i+2) = (u(i) - d(i)) * u(i+1)) * l(i)`

*x*vpoly	d,s	Vector polynomial evaluation	`call dvpoly(n,x,incx,m,c,incc,y,incy)`
			`y(i) = c(1) + c(2) * x(i) + ... +`
			`c(m) * x(i) ** (m-1)`