

ARGONNE NATIONAL LABORATORY  
9700 South Cass Avenue  
Argonne, IL 60439

---

ANL/MCS-TM-196

---

## **Fortran 77 Interface Specification to the SparsLinC 1.0 Library**

by

*Christian H. Bischof, Alan Carle,\* and Peyvand Khademi*

Mathematics and Computer Science Division

Technical Memorandum No. 196

May 1995

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Aerospace Agency under Purchase Order L25935D and Cooperative Agreement No. NCCW-0027; and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

ANL authors' email addresses: [bischof@mcs.anl.gov](mailto:bischof@mcs.anl.gov) [khademi@mcs.anl.gov](mailto:khademi@mcs.anl.gov)

\*Address: Center for Research on Parallel Computation, Rice University, 6100 S. Main St., Houston, TX 77005; email: [carle@cs.rice.edu](mailto:carle@cs.rice.edu).

# Contents

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Handling C Pointers from Fortran Programs</b>	<b>1</b>
2.1 Valid Pointer Values . . . . .	2
2.2 Initialization of C Data Structures . . . . .	2
2.3 Representation of Fortran Precisions in C . . . . .	2
2.4 Linking C and Fortran Modules . . . . .	3
2.5 Handling Mixed-Precision Codes . . . . .	3
<b>3 Uninitialized Vectors and Template Expansion</b>	<b>4</b>
<b>4 Naming Conventions</b>	<b>4</b>
<b>5 Representing Sparsity</b>	<b>4</b>
<b>6 Interface Routines</b>	<b>5</b>
6.1 Value Insertion and Extraction Routines . . . . .	5
6.2 Arithmetic Routines . . . . .	5
6.3 Conversion Routines . . . . .	6
6.4 Initialization, Configuration, and Inquiry Routines . . . . .	6
<b>Appendix: Detailed Interface Specification</b>	<b>7</b>
A.1 Value Insertion and Extraction Routines . . . . .	8
A.2 Arithmetic Routines . . . . .	14
A.3 Conversion Routines . . . . .	21
A.4 Initialization, Configuration, and Inquiry Routines . . . . .	25
<b>References</b>	<b>30</b>

# Fortran 77 Interface Specification to the SparsLinC 1.0 Library

by

*Christian H. Bischof, Alan Carle, and Peyvand Khademi*

## Abstract

The SparsLinC library, written in C, has been developed for exploiting sparsity in automatic differentiation of codes. Issues pertaining to the proper interface to the library from Fortran programs are discussed, including the interpretation of Fortran `INTEGER`s as C pointers, and the representation of Fortran precisions in C. The Appendix contains the full set of Fortran Interfaces to the SparsLinC library.

## 1 Introduction

A fundamental kernel in numerical linear algebra and also in automatic differentiation (see, e.g., [2]) is the computation of a linear combination of some vectors, namely,

$$w = \sum_{i=1}^k \alpha_i v_i,$$

where each  $\alpha_i$  is referred to as a “multiplier,”  $w$  as the “left-hand side vector,” and any of the  $v_i$ ’s as a “right-hand side vector.” Following Golub and Van Loan [4], we call this operation a **GAXPY**. In the cases of interest for automatic differentiation, the number  $k$  of vectors on the right-hand side is usually moderate, with  $k \leq 3$  forming the bulk of computations.

The SparsLinC (**S**parse **L**inear **C**ombinations) library has been developed to support this kernel computation for sparse vectors in `REAL`, `DOUBLE PRECISION`, `COMPLEX`, and `DOUBLE COMPLEX` arithmetic. A sparse vector contains a significant number of zero entries, and SparsLinC exploits this structure to save both on floating-point operations as well as on storage. SparsLinC employs a polyalgorithm in which a sparse vector is represented by one of three data structures, depending on the number and clustering of the indices corresponding to the nonzero entries in a vector. SparsLinC is mainly written in ANSI C with some Fortran 77 “wrapper” routines.

This document discusses how to access this library from a Fortran program and how to initialize and manipulate the C data structures that support sparse vectors from a Fortran program. Also discussed are the requirements on the Fortran implementation in this context.

## 2 Handling C Pointers from Fortran Programs

Since Fortran 77 does not have pointer variables, `INTEGER` variables are used to house the memory addresses of the C structures implementing sparse vectors. We adopt the convention that the Fortran `INTEGER` variable `VPTR` acts as a pointer to a sparse vector object, called `sparse_object(VPTR)`.

Table 1: Default Assumptions on Correspondence of Fortran and C Floating-Point Types

Fortran 77	C
REAL	float
DOUBLE PRECISION	double
COMPLEX	float [2]
DOUBLE COMPLEX	double [2]

## 2.1 Valid Pointer Values

We require that the Fortran **INTEGER** value “0” and the C pointer value “**NULL** ” are **identical**. This assumption is critical in deciding whether **VPTR** contains a valid address of a sparse derivative object. We assume that a **VPTR** of zero value implies that no sparse vector object has previously been associated with **VPTR** and that we must allocate one. Note, in particular, that a zero **VPTR** does *not* represent the sparse vector containing all zeros, although in “quiet” mode (see section 3) the correct representation for the vector of all zeros will be quietly allocated.

In our implementation a Fortran **INTEGER** representing a pointer to a sparse vector object can take the following values:

**0** : Uninitialized pointer to a sparse vector object.

**-1**: Special value denoting a sparse vector of all zeros in the [**REAL**, **DOUBLE PRECISION**] to [**COMPLEX**, **DOUBLE COMPLEX**] conversion routines (see sections 6.3 and 6.4).

**A valid pointer to a sparse data structure**: Such a valid address is assigned only by one of the routines in the SparsLinC library.

If one cannot rely on the fact that a positive value for **VPTR** contains a valid pointer to a sparse data structure, one must resort to memory authentication schemes to be able to answer this question (see, for example, [1, Problem 2.12]).

## 2.2 Initialization of C Data Structures

SparsLinC employs data structures that have to be initialized before any of the other SparsLinC routines can be called. The user must call the **XSPINI** routine to initialize these data structures.

## 2.3 Representation of Fortran Precisions in C

We make the default assumptions shown in Table 1 (which can be changed by redefining some macros) concerning the correspondence of C and Fortran data types. In particular, we assume that corresponding data types have the same word length and are aligned the same way. We further assume that for Fortran **COMPLEX** or **DOUBLE COMPLEX** variables, the first and second entries in the corresponding C **float** or **double** array of length two contain, respectively, the real and imaginary parts of a complex number.

## 2.4 Linking C and Fortran Modules

Two issues arise in the context of linking Fortran and C modules. One is the passing of strings between Fortran and C. Because this is notoriously difficult and nonuniform across different platforms, we avoid it. The only instance where we need to pass a string is for error-reporting purposes in the “verbose” routines (see section 3). These routines, as well as a few others, are provided as Fortran wrappers that perform the necessary string processing and then call the appropriate C routines.

The other issue is that of matching load module entry names generated by the C and Fortran compilers. For example, we must consider what case (upper or lower) entry names are supposed to be in or whether the Fortran compiler generates entry names with leading or trailing underscores. SparsLinC provides a macro expansion utility to easily address this issue when installing SparsLinC.

## 2.5 Handling Mixed-Precision Codes

All arithmetic routines are defined to handle the case where the multipliers and sparse vectors arguments are of the same type — any other use of the routines is wrong! Consequently, for each arithmetic computation — for example, GAXPY of arity 5 — four subroutines are provided (one for each of the precisions, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, and **DOUBLE COMPLEX**).

One way to handle mixed-precision computations (e.g., a GAXPY where some of the vectors and multipliers are stored in different precisions) is as follows:

1. Convert all multipliers to have the same precision as their corresponding vector, by using the Fortran conversion functions **REAL()**, **DBLE()**, **CMPLX()**, and **DCMPLX()**.
2. Accumulate all the vectors of the same type into temporary variables, by using the sparse arithmetic routines.
3. Convert all vectors to the “highest” precision, by using the sparse conversion routines. The usual hierarchy, in ascending order, is **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **DOUBLE COMPLEX**.
4. Accumulate these into a (possibly temporary) vector of that precision, by using the sparse arithmetic routines.
5. If necessary, truncate this vector to the precision desired for the left-hand side, by using the sparse conversion routines.

As an alternative, the following scheme has been suggested by Goldberg [3, p. 31]. Assume that an expression is represented as an expression tree, with the final result at its root. Then proceed as follows:

**Step 1:** Assign each operation a tentative precision, which is the maximum of its two operands, proceeding from the leaves to the root.

**Step 2:** Proceeding from the root to the leaves, assign to each operation the maximum of the tentative precision and the precision expected by the parent.

This would involve more conversion than the current rule, but could also easily be implemented. In any case, the responsibility for enforcing consistent input types to the sparse vector routines rests with the user of these routines.

### 3 Uninitialized Vectors and Template Expansion

During the execution of a program, we may try to access `sparse_object(VPTR)` for a `VPTR` that is `NULL`. Our routines check all `VPTR`’s corresponding to vectors on the right-hand side to see whether they are `NULL`. A `NULL` pointer indicates that the gradient that is being passed has not been initialized to point to a valid sparse representation of a gradient.

This case may happen, for example, if the code passed to the automatic differentiation tool contains an uninitialized variable `X` (perhaps arising from the fact that the user knows that his particular compiler blanks all variables before program execution). Then the occurrence of `X` on the right-hand side of an assignment statement may lead us to try to access `sparse_object(VPTR)` for a `VPTR` that is `NULL`, where `VPTR` is the pointer to the sparse derivative object associated with `X`.

All routines where the case of uninitialized right-hand sides can occur are provided in two versions:

**Verbose version:** Initialize `VPTR` to point to a representation of the vector of all zeros; report the file name, line number, and position of `VPTR` in the argument list to an error unit (default is `stdout`); and optionally halt the program. The routine `XSPUIV` can be called to customize these options. If a particular call refers to more than one `NULL` pointer, all occurrences of `NULL` pointers will be reported.

**Quiet version:** Initialize `VPTR` to point to a representation of the vector of all zeros, and proceed quietly.

### 4 Naming Conventions

We adopt the following naming conventions for SparsLinC routines:

- The first letter will be an “S”, “D”, “C”, “Z”, or “X,” indicating, respectively, whether the routine manipulates vectors in `REAL`, `DOUBLE PRECISION`, `COMPLEX`, and `DOUBLE COMPLEX` precision or whether it is a nonnumeric utility routine.
- The second and third letters will be “SP”, to denote that the routine is in the `SparsLinC` library.
- For routines that may encounter uninitialized right-hand sides and are provided in a verbose and quiet version, the fourth and fifth letter are an abbreviation of the task performed by the routine, and the sixth letter will be a “V” or “Q,” respectively. For other routines, the last two or three letters will be an abbreviation of the task performed by the routine.

Unless otherwise specified, an identifier ending in “PTR” refers to an `INTEGER` variable containing the pointer to a sparse derivative object. By virtue of the side effects associated with the handling of uninitialized variables, derivative objects corresponding to entries on the right-hand side of a `GAXPY` may be modified in such a call (from zero to a valid pointer).

### 5 Representing Sparsity

A sparse vector with *nonz* nonzeros can be represented in Fortran by an `INTEGER` array of length *nonz* containing the indices of nonzero entries, and a floating-point array of appropriate precision containing the corresponding values. For example, the 7-vector

(11.0, 0, 33.0, 44.0, 0, 0, 77.0)

could have the following sparse representation using two arrays of length 4 each:

Index Array:	1	3	4	7
Value Array:	11.0	33.0	44.0	77.0

We will refer to this two-array representation of the vector as the **Fortran Sparse Format**. The corresponding nonsparse representation, which we will call the **Fortran Nonsparse Format**, would be a floating-point array of length 7, containing zeros in entries 2, 5, and 6. Lastly, there is the **SparsLinC Sparse Format**, which is the internal SparsLinC representation of the vector.

## 6 Interface Routines

The following sections give an overview of the functionality provided by SparsLinC. A complete description is provided in the appendix.

### 6.1 Value Insertion and Extraction Routines

We provide the following routines to insert/extract values into/from the sparse vector representations:

[S,D,C,Z]**SPSD**: Convert a sparse vector stored in Fortran Sparse Format into the SparsLinC Sparse Format vector (used for initializing the **SeeD** matrix).

[S,D,C,Z]**SPXD**[Q,V]: **EX**tract a SparsLinC Sparse Format vector into a Fortran Nonsparse (**D**ense) Format vector.

[S,D,C,Z]**SPXS**[Q,V]: **EX**tract a SparsLinC Sparse Format vector into a Fortran Sparse Format vector.

[S,D,C,Z]**SPXM**[Q,V]: **EX**tract a SparsLinC Sparse Format vector, **M**ultiply it by a scalar and add the result to a Fortran Nonsparse Format vector.

[S,D,C,Z]**SPXA**[Q,V]: **EX**tract and **A**dd a SparsLinC Sparse Format vector to a Fortran Nonsparse Format vector.

[S,D,C,Z]**SPPR**[Q,V]: **PR**int a sparse vector.

### 6.2 Arithmetic Routines

[S,D,C,Z]**SPCP**[Q,V] : **CoPy** a vector.

[S,D,C,Z]**SPZRO**: Assign the vector of all **ZeRO**s to a sparse vector.

[S,D,C,Z]**SPVZR**: Assign to each entry in an array of sparse vectors the **V**ector of all **ZeRO**s.

[S,D,C,Z]**SPG1**[Q,V], ..., [S,D,C,Z]**SPG5**[Q,V]: Perform a **GAXPY** with **1** to **5** vectors.

[S,D,C,Z]**SPGX**[Q,V]: Perform a **GAXPY** with more than 5 vectors. Unlike the “special-case” GAXPY implementations, this routine assumes that pointers to right-hand-side vectors as well as multipliers are packed into a vector. The particular choice of 5 for the cutoff was motivated, on the one hand, by the fact that in our experience the great majority of GAXPY’s occurring

in the automatic differentiation context involve no more than five vectors and, on the other hand, by the fact that every special GAXPY implementation adds eight new entries to the (already rather large) library.

**[C,Z]SPIM[Q,V]:** Extract the **IM**aginary part of a **COMPLEX** or **DOUBLE COMPLEX** sparse vector into a **REAL** or **DOUBLE PRECISION** sparse vector. Corresponds to **IMAG()**.

**[C,Z]SPCJ[Q,V]:** **ConJ**ugate a **COMPLEX** or **DOUBLE COMPLEX** sparse vector. Corresponds to **CONJG()**.

### 6.3 Conversion Routines

**[S,D,C,Z]SP2S[Q,V]:** Sparse vector conversion to **REAL** (Single Precision). **REAL** or **DOUBLE PRECISION** conversion to **REAL**, or extraction of real part of **COMPLEX** or **DOUBLE COMPLEX** sparse vector into a **REAL** sparse vector. Corresponds to **REAL()**.

**[S,D,C,Z]SP2D[Q,V]:** Sparse vector conversion to **DOUBLE PRECISION**. **REAL** or **DOUBLE PRECISION** conversion to **DOUBLE PRECISION**, or extraction of real part of **COMPLEX** or **DOUBLE COMPLEX** sparse vector into a **DOUBLE PRECISION** sparse vector. Corresponds to **DBLE()**.

**[S,D]SP2C[Q,V]:** Sparse vector conversion of **[REAL, DOUBLE PRECISION]** to **COMPLEX**. The **CMPLX()** Fortran intrinsic can take one or two arguments. We adopt the convention that if **VPTR** equals -1, then the corresponding vector is taken to be the zero vector. We use a value other than 0 to distinguish this case from the one where a vector is uninitialized.

**[S,D]SP2Z[Q,V]:** Sparse vector conversion of **[REAL, DOUBLE PRECISION]** to **DOUBLE COMPLEX** (**Z**). Various Fortran vendor compilers support the **REAL** to **DOUBLE COMPLEX** conversion, usually by adding an intrinsic **DCMPLX()** or **ZCMPLX()**. Hence, we also provide the equivalent vector conversion, for completeness. We adopt the convention that if **VPTR** equals -1, then the corresponding vector is taken to be the zero vector. We use a value other than 0 to distinguish this case from the one where a vector is uninitialized.

### 6.4 Initialization, Configuration, and Inquiry Routines

**XSPINI:** **INI**tialize C data structures. Must be called before calling the derivative code employing other SparsLinC library calls, and must be called only once. When called more than once, all but the first call act as no-ops.

**XSPCNF:** **CoN**figure certain internal SparsLinC parameters.

**XSPUIV:** Configure action to be taken upon encountering **Un**initialized **V**ariables.

**XSPMEM:** Report amount of **MEM**ory used for representing sparse vectors.

**XSPFRA:** **F**ree All memory for C sparse vector data structures. After a call to this routine, **all VPTR's are left dangling**. The purpose of this routine is to free memory when derivative computation is completed.



## Appendix: Detailed Interface Specification

To allow for detailed error reporting when encountering uninitialized sparse derivative objects, we provide a template expansion mechanism that maps calls to templates into calls to actual routines (which may or may not pass line number and file name), in the fashion outlined in section 3. Routines for which such a functionality is provided have the optional arguments [line,file] in their parameter list, where the variable “line” is declared as `INTEGER` and “file” as `CHARACTER*(*)` , and both are input parameters. Additionally, for these routines (such as the dense extraction routine which we use here as a prototypical example), we use the following notation in the header of the routine description:

```
SUBROUTINE SSPXD[Q,V] (XVEC, INLEN, VPTR, OUTLEN, INFO, [line, file])
```

as a shorthand for the following:

```
SUBROUTINE SSPXDQ (XVEC, INLEN, VPTR, OUTLEN, INFO)
SUBROUTINE SSPXDV (XVEC, INLEN, VPTR, OUTLEN, INFO, line, file)
```

We also utilize variable names with up to eight characters, although all subroutine names are no longer than six characters.

Again, we adopt the term `sparse_object(VPTR)` as a shorthand for “the `sparse_object( )` pointed to by `VPTR`.” To save space, we provide only the calling sequence for one particular floating-point precision.

## A.1 Value Insertion and Extraction Routines

---

### SSPSD, DSPSD, CSPSD, ZSPSD

SUBROUTINE SSPSD (VPTR, INDVEC, VALVEC, LEN)

#### Purpose

Conversion of a vector in Fortran Sparse Format into a vector in SparsLinC Sparse Format. The Fortran Sparse Format vector is given by the two arrays, `INDVEC(1:LEN)` and `VALVEC(1:LEN)`, representing the indices and values of a sparse vector  $x$  (say), respectively.  $x$  is copied into `sparse_object(VPTR)` which is the vector in SparsLinC Sparse Format. The indices in `INDVEC` need not be in any particular order (internally, SSPSD performs an ascending order sort). However, `INDVEC` and `VALVEC` must be identically aligned, i.e., if in the Fortran Nonsparse Format  $x$  has a nonzero entry at index  $i$  with value  $v$ , then for some  $J$ , `INDVEC(J) = i` and `VALVEC(J) = v`. SSPSD performs a *destructive copy*, i.e., if `sparse_object(VPTR)` had been previously allocated (via SSPSD or as a result of being an output argument of some other SparsLinC routine), the previous information in `sparse_object(VPTR)` is lost, and the dynamically-allocated memory where that information resided is deallocated.

#### Arguments

VPTR	(output) INTEGER Upon exit, <code>sparse_object(VPTR)</code> contains a copy of the sparse vector represented by <code>INDVEC</code> and <code>VALVEC</code> .
INDVEC	(input) INTEGER array, dimension (LEN) Indices of the nonzero values of the sparse vector. (We assume that indices are $\geq 1$ , therefore, <code>INDVEC</code> entries $\leq 0$ would be incorrect and would result in a runtime error.)
VALVEC	(input) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (LEN) Nonzero values of the sparse vector.
LEN	(input) INTEGER $LEN \geq 0$ is the number of nonzeros in the sparse vector. If <code>LEN</code> equals zero, <code>VPTR</code> is initialized to point to the vector of all zeros and <code>INDVEC</code> and <code>VALVEC</code> are not referenced.

---

---

## SSPXD[Q,V], DSPXD[Q,V], CSPXD[Q,V], ZSPXD[Q,V]

SUBROUTINE SSPXD[Q,V] (XVEC, INLEN, VPTR, OUTLEN, INFO, [line, file])

### Purpose

Extracts `sparse_object(VPTR)` into the Fortran Nonsparse Format vector `XVEC`.

### Arguments

<b>XVEC</b>	(output) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (INLEN) On exit, if <b>INFO</b> equals zero, <code>XVEC(1:INLEN)</code> will contain a dense representation of <code>sparse_object(VPTR)</code> . If <b>OUTLEN</b> is less than <b>INLEN</b> then <code>XVEC(OUTLEN+1:INLEN)</code> is initialized to all zeros. If <b>INFO</b> $\neq$ 0, <code>XVEC</code> is not referenced.
<b>INLEN</b>	(input) INTEGER Length of <code>XVEC</code> .
<b>VPTR</b>	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If <b>VPTR</b> equals <code>NULL</code> , it is initialized to point to the vector of all zeros (which is why it might be an output argument).
<b>OUTLEN</b>	(output) INTEGER Largest index in the nonzero index set in <code>sparse_object(VPTR)</code> . This value will always be returned, whether <code>XVEC</code> is initialized or not. See the description of <b>INFO</b> below.
<b>INFO</b>	(output) INTEGER If <b>INLEN</b> $<$ <b>OUTLEN</b> , <b>INFO</b> will be set to -1, and <code>XVEC</code> is not referenced. Otherwise, <b>INFO</b> is set to 0, and <code>XVEC(1:INLEN)</code> is initialized to a Fortran Nonsparse Format copy of <code>sparse_object(VPTR)</code> .

---

---

## SSPXS[Q,V], DSPXS[Q,V], CSPXS[Q,V], ZSPXS[Q,V]

SUBROUTINE SSPXS[Q,V] (INDVEC, VALVEC, INLEN, VPTR, OUTLEN, INFO, [line, file])

### Purpose

Extracts `sparse_object(VPTR)` into the Fortran Sparse Format vector represented by the two arrays, `INDVEC` and `VALVEC`.

### Arguments

INDVEC	(output) INTEGER array, dimension (INLEN) On exit, if INFO equals zero, <code>INDVEC(1:OUTLEN)</code> contains the indices of the nonzero entries of <code>sparse_object(VPTR)</code> . If <code>INFO &lt;&gt; 0</code> , <code>INDVEC</code> is not referenced. If <code>INFO = 0</code> and <code>OUTLEN</code> is less than <code>INLEN</code> then <code>INDVEC(OUTLEN+1:INLEN)</code> is not referenced.
VALVEC	(output) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (INLEN) On exit, if INFO equals zero, <code>VALVEC(1:OUTLEN)</code> will contain the nonzero entries of <code>sparse_object(VPTR)</code> . If <code>INFO &lt;&gt; 0</code> , <code>VALVEC</code> is not referenced. If <code>INFO = 0</code> and <code>OUTLEN</code> is less than <code>INLEN</code> then <code>VALVEC(OUTLEN+1:INLEN)</code> is not referenced.
INLEN	(input) INTEGER Length of <code>INDVEC</code> and <code>VALVEC</code> .
VPTR	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If <code>VPTR</code> equals <code>NULL</code> , it is initialized to point to the vector of all zeros (which is why it might be an output argument).
OUTLEN	(output) INTEGER Number of nonzeros in <code>sparse_object(VPTR)</code> . This value will always be returned, whether <code>INDVEC</code> and <code>VALVEC</code> are initialized or not. See the description of <code>INFO</code> below.
INFO	(output) INTEGER If <code>INLEN &lt; OUTLEN</code> , <code>INFO</code> will be set to -1, and <code>INDVEC</code> and <code>VALVEC</code> are not referenced. Otherwise, <code>INFO</code> is set to 0, and <code>INDVEC(1:OUTLEN)</code> and <code>VALVEC(1:OUTLEN)</code> are initialized to the Fortran Sparse Format copy of <code>sparse_object(VPTR)</code> .

---

---

## SSPXM[Q,V], DSPXM[Q,V], CSPXM[Q,V], ZSPXM[Q,V]

SUBROUTINE SSPXM[Q,V] (XVEC, INLEN, MULT, VPTR, OUTLEN, INFO, [line, file])

### Purpose

Adds the weighted contents of `sparse_object(VPTR)` to the Fortran Nonsparse Format vector `XVEC`, where `MULT` is the multiplicative weight (i.e.,  $XVEC = XVEC + MULT * \text{sparse\_object}(VPTR)$ ). For example, say `XVEC` is a vector of length 7 containing all ones, `MULT` is equal to 2.0, and `sparse_object(VPTR)` is as follows:

Index Array:	1	3	4	7
Value Array:	11.0	33.0	44.0	77.0

Subsequent to the call to this routine, `XVEC` would contain the following:

(23.0, 1.0, 67.0, 89.0, 1.0, 1.0, 154.0)

### Arguments

<b>XVEC</b>	(input/output) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (INLEN) On exit, if <code>INFO</code> equals zero, <code>XVEC(1:INLEN)</code> will have added to it the weighted contributions of the values in <code>sparse_object(VPTR)</code> , with <code>MULT</code> specifying the weight. If <code>INFO &lt;&gt; 0</code> , <code>XVEC</code> is not modified.
<b>INLEN</b>	(input) INTEGER Length of <code>XVEC</code> .
<b>MULT</b>	(input) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] Multiplier.
<b>VPTR</b>	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If <code>VPTR</code> equals <code>NULL</code> , it is initialized to point to the vector of all zeros (which is why it might be an output argument).
<b>OUTLEN</b>	(output) INTEGER Largest index in the nonzero index set in <code>sparse_object(VPTR)</code> . This value will always be returned, whether <code>XVEC</code> is modified or not. See the description of <code>INFO</code> below.
<b>INFO</b>	(output) INTEGER If <code>INLEN &lt; OUTLEN</code> , <code>INFO</code> will be set to -1, and <code>XVEC</code> is not modified. Otherwise, <code>INFO</code> is set to 0, and <code>XVEC(1:INLEN)</code> is modified as described above.

---

---

## SSPXA[Q,V], DSPXA[Q,V], CSPXA[Q,V], ZSPXA[Q,V]

SUBROUTINE SSPXA[Q,V] (XVEC, INLEN, VPTR, OUTLEN, INFO, [line, file])

### Purpose

Adds the contents of `sparse_object(VPTR)` to the Fortran Nonsparse Format vector `XVEC` (i.e.,  $XVEC = XVEC + \text{sparse\_object}(VPTR)$ ). (SPXA is identical to the SPXM routine with `MULT` equal to one; see the documentation for SPXM.)

### Arguments

<b>XVEC</b>	(input/output) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (INLEN) On exit, if <code>INFO</code> equals zero, <code>XVEC(1:INLEN)</code> will have added to it the values in <code>sparse_object(VPTR)</code> . If <code>INFO &lt; 0</code> , <code>XVEC</code> is not modified.
<b>INLEN</b>	(input) INTEGER Length of <code>XVEC</code> .
<b>VPTR</b>	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If <code>VPTR</code> equals <code>NUL</code> , it is initialized to point to the vector of all zeros (which is why it might be an output argument).
<b>OUTLEN</b>	(output) INTEGER Largest index in the nonzero index set in <code>sparse_object(VPTR)</code> . This value will always be returned, whether <code>XVEC</code> is modified or not. See the description of <code>INFO</code> below.
<b>INFO</b>	(output) INTEGER If <code>INLEN &lt; OUTLEN</code> , <code>INFO</code> will be set to -1, and <code>XVEC</code> is not modified. Otherwise, <code>INFO</code> is set to 0, and <code>XVEC(1:INLEN)</code> is modified as described above.

---

---

## SSPPR[Q,V], DSPPR[Q,V], CSPPR[Q,V], ZSPPR[Q,V]

SUBROUTINE SSPPR[Q,V] (VPTR, EXT, [line, file])

### Purpose

Writes number of nonzeros as well as index/value pairs of sparse\_object(VPTR) onto stdout or a file, with the following format:

```
Number of nonzeros = . . .
      Index      Value
      -----
      . . .      . . .
      . . .      . . .
```

### Arguments

VPTR	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If VPTR is NULL, it is initialized to point to the vector of all zeros (which is why it might be an output argument).
EXT	(input) INTEGER Must be in the range [0,999]. If EXT equals zero, output written is to stdout. Otherwise EXT is converted to its ASCII equivalent and used as the extension appended to the filename "SPPR." and output is written to this file.

---

## A.2 Arithmetic Routines

---

### SSPCP[Q,V], DSPCP[Q,V], CSPCP[Q,V], ZSPCP[Q,V]

SUBROUTINE SSPCP[Q,V] (DESTPTR, SRCPTR, [line, file])

#### Purpose

Copies `sparse_object(SRCPTR)` into `sparse_object(DESTPTR)`.

#### Arguments

DESTPTR        (output) INTEGER  
                 Pointer to sparse vector object.

SRCPTR        (input/output) INTEGER  
                 Pointer to sparse vector object.

---



---

## SSPZRO, DSPZRO, CSPZRO, ZSPZRO

SUBROUTINE SSPZRO (VPTR)

### Purpose

Initializes `sparse_object(VPTR)` to the vector of all zeros.

### Arguments

VPTR	(input/output) INTEGER
	Pointer to sparse vector object.

---

## SSPVZO, DSPVZO, CSPVZO, ZSPVZO

SUBROUTINE SSPVZO (VPTRS, n)

### Purpose

Initializes `sparse_object(VPTRS(i))` to point to the vector of all zeros for  $i = 1, \dots, N$ .

### Arguments

VPTRS	(input/output) INTEGER array, length (N) Array of pointers to sparse vector objects.
N	(input) INTEGER Length of VPTRS array.

---

---

SSPG1[Q,V], ..., SSPG5[Q,V], DSPG1[Q,V], ..., DSPG5[Q,V],  
CSPG1[Q,V], ..., CSPG5[Q,V], ZSPG1[Q,V], ..., ZSPG5[Q,V]

SUBROUTINE SSPG1[Q,V](DESTPTR, ALPHA1, V1PTR, [line, file])

....

SUBROUTINE SSPG5[Q,V](DESTPTR, ALPHA1, V1PTR, ..., ALFA5, V5PTR, [line, file])

## Purpose

Computes

$$\text{sparse\_object}(\text{DESTPTR}) = \sum_{i=1}^k \text{ALPHA}_i * \text{sparse\_object}(\text{ViPTR})$$

for values of  $k$  from 1 to 5. It is assumed that all  $\text{ViPTR}$  are pointers to sparse vector objects representing the same precision (REAL, DOUBLE PRECISION, COMPLEX, or DOUBLE COMPLEX) and that all the multipliers  $\text{ALPHA}_i$  are of the same precision as well.

## Arguments

DESTPTR (output) INTEGER

Pointer to sparse vector object.

ALPHA $_i$  (input) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] .

Multipliers of  $\text{sparse\_object}(\text{ViPTR})$ .

ViPTR (input/output) INTEGER

Pointers to sparse vector object.

---

---

## SSPGX[Q,V], DSPGX[Q,V], CSPGX[Q,V], ZSPGX[Q,V]

SUBROUTINE SSPGX[Q,V] (DESTPTR, ARITY, ALPHAVEC, VPTRVEC, [line, file])

### Purpose

Computes

$$\text{sparse\_object}(\text{DESTPTR}) = \sum_{i=1}^{\text{ARITY}} \text{ALPHAVEC}[i] * \text{sparse\_object}(\text{VPTRVEC}[i]).$$

It is assumed that all VPTRVEC[i] are pointers to sparse vector objects representing the same precision (REAL, DOUBLE PRECISION, COMPLEX, or DOUBLE COMPLEX) and that all the multipliers ALPHAVEC[i] are of the same precision as well.

### Arguments

DESTPTR	(output) INTEGER Pointer to sparse vector object.
ARITY	(input) INTEGER Number of sparse derivative objects on the right-hand side.
ALPHAVEC	(input) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (ARITY) Scalar multipliers.
VPTRVEC	(input/output) INTEGER array, dimension (ARITY). Array of pointers to sparse vector objects.

---

---

## CSPIM[Q,V], ZSPIM[Q,V]

SUBROUTINE CSPIM[Q,V] (DESTPTR,VPTR, [line, file])

### Purpose

Returns the imaginary part of `sparse_object(VPTR)` in `sparse_object(DESTPTR)`.

**CSPIM:** `sparse_object(VPTR)` is assumed to be in `COMPLEX` format and `sparse_object(DESTPTR)` will be in `REAL` format.

**ZSPIM:** `sparse_object(VPTR)` is assumed to be in `DOUBLE COMPLEX` format and `sparse_object(DESTPTR)` will be in `DOUBLE PRECISION` format.

### Arguments

DESTPTR	(output) INTEGER Pointer to sparse vector object in <code>REAL [DOUBLE PRECISION]</code> format.
VPTR	(input/output) INTEGER Pointer to sparse vector object in <code>COMPLEX [DOUBLE COMPLEX]</code> format.

---

---

## CSPCJ[Q,V], ZSPCJ[Q,V]

SUBROUTINE CSPCJ[Q,V] (DESTPTR,VPTR, [line, file])

### Purpose

Returns the conjugate complex of sparse\_object(VPTR) in sparse\_object(DESTPTR).

### Arguments

DESTPTR	(output) INTEGER Pointer to sparse vector object in COMPLEX [DOUBLE COMPLEX] format.
VPTR	(input/output) INTEGER Pointer to sparse vector object in COMPLEX [DOUBLE COMPLEX] format.

---

### A.3 Conversion Routines

---

#### SSP2S[Q,V], DSP2S[Q,V], CSP2S[Q,V], ZSP2S[Q,V]

SUBROUTINE SSP2S[Q,V] (DESPTR, VPTR, [line, file])

#### Purpose

**SSP2S:** `sparse_object(VPTR)` is copied to `sparse_object(DESPTR)`.

**DSP2S:** A copy of `sparse_object(VPTR)` is truncated to REAL format and copied into `sparse_object(DESPTR)`.

**CSP2S:** The real part of `sparse_object(VPTR)` is copied into `sparse_object(DESPTR)`.

**ZSP2S:** A copy of the real part of `sparse_object(VPTR)` is truncated to REAL format and copied into `sparse_object(DESPTR)`.

#### Arguments

DESPTR	(output) INTEGER Pointer to sparse vector object in REAL format.
VPTR	(input/output) INTEGER Pointer to sparse vector object in REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] format.

---

---

## SSP2D[Q,V], DSP2D[Q,V], CSP2D[Q,V], ZSP2D[Q,V]

SUBROUTINE SSP2D[Q,V] (DESTPTR, VPTR, [line, file])

### Purpose

**SSP2D:** A copy of `sparse_object(VPTR)` is converted to `DOUBLE PRECISION` format and copied to `sparse_object(DESTPTR)`.

**DSP2D:** `sparse_object(VPTR)` is copied into `sparse_object(DESTPTR)`.

**CSP2D:** A copy of the real part of `sparse_object(VPTR)` is converted to `DOUBLE PRECISION` format and copied into `sparse_object(DESTPTR)`.

**ZSP2D:** The real part of `sparse_object(VPTR)` is copied into `sparse_object(DESTPTR)`.

### Arguments

DESTPTR	(output) INTEGER Pointer to sparse vector object in <code>DOUBLE PRECISION</code> format.
VPTR	(input/output) INTEGER Pointer to sparse vector object in <code>REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX]</code> format.

---



---

## SSP2C[Q,V], DSP2C[Q,V]

SUBROUTINE SSP2C[Q,V] (DESTPTR, VRLPTR, VIMPTR, [line, file])

### Purpose

sparse\_object(VRLPTR) and sparse\_object(VIMPTR) correspond to the real and imaginary part of the complex vector to be built. It is assumed that sparse\_object(VRLPTR) and sparse\_object(VIMPTR) are stored in the same format (REAL or DOUBLE PRECISION).

**SSP2C:** sparse\_object(DESTPTR) is assigned to have the values of sparse\_object(VRLPTR) as its real part, and the values of sparse\_object(VIMPTR) as its imaginary part.

**DSP2C:** Copies of sparse\_object(VRLPTR) and sparse\_object(VIMPTR) are truncated to REAL format, and these values are assigned to sparse\_object(DESTPTR) as the real and imaginary parts, respectively.

### Arguments

DESTPTR	(output) INTEGER Pointer to sparse vector object in COMPLEX format.
VRLPTR	(input/output) INTEGER Pointer to sparse vector object in REAL [DOUBLE PRECISION] format. The case of VRLPTR equal -1 is treated as if sparse_object(VRLPTR) was the vector of all zeros but is not considered an occurrence of an uninitialized pointer.
VIMPTR	(input/output) INTEGER Pointer to sparse vector object in REAL [DOUBLE PRECISION] format. The case of VIMPTR equal -1 is treated as if sparse_object(VIMPTR) was the vector of all zeros but is not considered an occurrence of an uninitialized pointer.

---

---

## SSP2Z[Q,V], DSP2Z[Q,V]

SUBROUTINE SSP2Z[Q,V] (DESTPTR, VRLPTR, VIMPTR, [line, file])

### Purpose

sparse\_object(VRLPTR) and sparse\_object(VIMPTR) correspond to the real and imaginary part of the complex vector to be built. It is assumed that sparse\_object(VRLPTR) and sparse\_object(VIMPTR) are stored in the same format (REAL or DOUBLE PRECISION).

**SSP2Z:** Copies of sparse\_object(VRLPTR) and sparse\_object(VIMPTR) are converted to DOUBLE PRECISION format, and these values are assigned to sparse\_object(DESTPTR) as real and imaginary parts, respectively.

**DSP2Z:** sparse\_object(DESTPTR) is assigned to have the values of sparse\_object(VRLPTR) as its real part and the values of sparse\_object(VIMPTR) as its imaginary part.

### Arguments

DESTPTR	(output) INTEGER Pointer to sparse vector object in DOUBLE COMPLEX format.
VRLPTR	(input/output) INTEGER Pointer to sparse vector object in REAL [DOUBLE PRECISION] format. The case of VRLPTR equal -1 is treated as if sparse_object(VRLPTR) was the vector of all zeros but is not considered an occurrence of an uninitialized pointer.
VIMPTR	(input/output) INTEGER Pointer to sparse vector object in REAL [DOUBLE PRECISION] format. The case of VIMPTR equal -1 is treated as if sparse_object(VIMPTR) was the vector of all zeros but is not considered an occurrence of an uninitialized pointer.

---

## A.4 Initialization, Configuration, and Inquiry Routines

---

### XSPINI

SUBROUTINE XSPINI

#### Purpose

Initializes the sparse data structures by dynamically allocating memory for some SparsLinC-internal global variables. It must be called before any of the other SparsLinC routines (except for calls to XSPCNF with OPTs 1-15) and needs to be called no more than once (when called more than once, all but the first call act as no-ops).

#### Arguments

none

---

---

## XSPCNF

SUBROUTINE XSPCNF (OPT, VAL)

### Purpose

Allows user to customize SparsLinC for each run. The following table specifies for each parameter its name, option number, default value, and range of allowable values. “SSbucket\_size” and “CSbucket\_size” are the number of entries per array in the linked list representation of a single-subscript and compressed-subscript vector respectively. “switch\_threshold” is the number of nonzero entries from which on a SparsLinC sparse vector is represented in compressed-subscript form. A more detailed explanation of this issue is provided in Appendix B of the ADIFOR 2.0 User’s Guide, Section B.4.3.

<u>Name</u>	<u>OPT</u>	<u>Default</u>	<u>Range</u>
SSbucket_size	1	8	>1
CSbucket_size	2	32	>1
switch_threshold	3	16	>1

XSPCNF with OPT = 1 or OPT = 2 may be called only before calling XSPINI. Calling XSPCNF with OPT = 1 or 2 after a call to XSPINI will result in a runtime error. Calls to XSPCNF with OPT = 3 can be made at any time.

### Arguments

OPT	(input) INTEGER Specifies the Option number associated with a given parameter as given in the above table.
VAL	(input) INTEGER The new value for the parameter specified by OPT.

---

---

## XSPUIV

SUBROUTINE XSPUIV (ACTION, VALUE)

### Purpose

Configures handling of uninitialized vectors. By default, if XSPUIV is not called, all error messages are written to standard output, and program execution continues after encountering an uninitialized right-hand side.

### Arguments

ACTION (input) INTEGER

ACTION = 1: Specifies the unit number for error reporting.

ACTION = 2: Specifies the maximum number of errors to be reported.

ACTION = 3: Specifies whether program should continue or abort.

VALUE (input) INTEGER

If ACTION = 1, VALUE specifies the unit number for error reporting.

If ACTION = 2,

- VALUE = -1 indicates that all errors are to be reported,
- VALUE = 0 indicates that no errors are to be reported, and
- VALUE =  $k > 0$  indicates that at most  $k$  errors are to be reported.

If ACTION = 3,

- VALUE = 0 indicates that program execution should continue,
  - VALUE = 1 indicates that the program should halt upon encountering the first uninitialized variable, and
  - VALUE = 2 indicates that the program should halt after printing the maximum number of error messages.
-

---

## XSPMEM

SUBROUTINE XSPMEM (USEDKB)

### Purpose

Reports how many Kbytes have been allocated in SparsLinC.

### Arguments

USEDKB            (output) REAL .  
The number of KBytes of storage allocated for SparsLinC data structures.

---

---

## XSPFRA

SUBROUTINE XSPFRA

### Purpose

Frees all memory allocated for C sparse vector data structures. **Note: all pointers to sparse directional gradient variables (VPTR's) are left dangling.**

### Arguments

none

---

## References

- [1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
- [2] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.
- [3] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
- [4] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.