Stanford Exploration Project, Report 92, November 12, 1997, pages 357–??

Stanford Exploration Project, Report 92, November 12, 1997, pages 357–??

Fortran90: Introduction and use in 3-D geophysical problems

Robert G. Clapp and Sean Crawley[1]

keywords: *algorithm, arrays, three-dimensional, processing*

ABSTRACT

There is a general consensus in the scientific community that object oriented programming is the correct way to perform research. The object oriented approach allows much more complex ideas to be explored by allowing the scientist to concentrate on ideas rather than algorithms. This feature is especially attractive in problems such as we encounter in 3-D processing, where dealing with the huge data size and irregularity is extremely cumbersome and prone to errors, in traditional languages such as Fortran77. We use Fortran90, built upon the existing SEP90 accessors routines, as the building blocks to an object oriented processing environment. Our initial efforts indicate that a Fortran90 base can provide the flexibility of C++, while maintaining the simplicity of Fortran, to effectively solve complex geophysical problems.

INTRODUCTION

In the last several years numerous attempts have been made to move geophysical processing into an object oriented structure. There have been some successes, notably the Rice Inversion Project. SEP also has attempted to move towards an object oriented programming style ((1993),(1994)), but these efforts have met with only limited success. There are three notable reasons why SEP's attempts did not reach fruition. First, C++, the object oriented language of choice, has a steep learning curve, which is especially problematic since SEPers have traditionally been Fortran 77 programmers. Second C++ contains pointers that make successful parallel programming difficult. And finally it is only recently that SEP has moved into 3-D processing where the benefits of the object oriented approach are magnified. The introduction of SEP90(Biondi et al., 1996) and loan from SGI of a 16 node power challenge has made the time ripe for a new venture into object oriented programming using Fortran90. Fortran90, offers an attractive alternative that addresses many of

[1]**email:** bob@sep.stanford.edu,sean@sep.stanford.edu

the shortcomings of C++. It does not suffer from the pointer problem of C++, so is easily parallelizable. Finally, Fortran90's basic structure is very similar to Fortran 77 which makes it easily accessible to all of SEP. What we propose is a Fortran90 framework that mirrors the general data structure of SEPlib90, but provides ease of use and freedom from much of the overhead coding that SEPlib90's relatively complex data structure necessitates.

The paper will be broken up into five parts. The first portion will cover some of the basic features of Fortran90 and our reason for choosing it at as our object oriented language. We will then look into our SEP data structure and our routines to create and destroy them. The next section will cover data I/O and our accessors. The fourth portion will describe our philosophy on operators and solvers, and some of the operators that we have currently implemented. We will conclude with our future plans for an object oriented programming environment and our conclusions on its usefulness.

WHY FORTRAN90?

Since 1989 there have been several major attempts to start C++ object oriented projects at SEP, all have met with only limited success and died off within a year of their conception. This in spite of having some of the best minds at SEP behind them. The reason for each projects' failure vary, but two general problems undermined each attempt.

Parallel Computing

SEP students have always pushed the limits of their available computing resources. From 1991-95 the "limit" was our 32-node CM5 Connection Machine. Since November 1995, the limit has been set by a 16-node SGI Power Challenge. The CM5 parallelization was most easily utilized with CM Fortran. For the SGI parallelization is possible in C, and therefore C++, but automatic parallelization is extremely limited due to the way pointers are handled in each language.

As a result even the most enthusiastic supporters of C++ abandoned it when it came to serious processing of real data. With SEP's shift into 3-D, the importance of doing high performance parallel computing is even greater.

Language Barrier

The SEPlib base is written in C, but historically people at SEP have done most of their coding in Fortran. Most students program exclusively in one language and choose that language early at SEP. Fortran is the most common choice because

- all of the code in Jon's books is written in the Fortran dialect Ratfor and
- it is the language most students know when they arrive at SEP.

Once a student feels comfortable in a language he is apt not to change, especially from Fortran to C or C++ where many concepts are *extremely* foreign. As a result SEP's C++ projects were inaccessible to large portion of the students and these students ideas, operators, and solvers could not be contributed to the project.

Evolution of Fortran

Fortran90 has certain powerful features, which fill traditional gaps in Fortran programming. The lack of dynamically allocatable memory, which has spawned a number of workarounds, is no longer a problem. User-defined structures, long available to C programmers (and a source of jealously for Fortran programmers) and operator overloading, available in C++, are now available as well. Unlike C, however, Fortran90 maintains Fortran's limited definition of pointers, its complex valued variables, etc., which serve to make coding more straightforward. Fortran90 also has a number of useful array manipulation intrinsics (matrix multiplications, transposes, etc.) which further serve to make code simple, and clean.

Fortran90 implements its new features chiefly through the unifying concept of the module. Modules provide what is termed an *explicit interface* between programs, subroutines, data structures, etc. What this means is that it enables the compiler to explicitly check the types and dimensionality of arguments as they are passed from place to place, and thus to resolve overloaded operations and so forth. More importantly, from the user's point of view, modules and their explicit interfaces provide a single conceptual basis for the advanced programming tools available in the language.

FORTRAN90 BASICS

This is not the right venue for a Fortran90 textbook. However, it is worth illustrating certain important features that Fortran90 has and that we make extensive use of in our data structure, and it is also worth displaying the easy and straightforward coding style that Fortran90 encourages.

Dynamic Allocation

There are two main types of allocatable arrays available in Fortran90, the `allocatable` and `pointer` arrays. `Allocatable` arrays must be allocated in the main program, and so while they may be passed back and forth to subroutines, they may

not be function results. Giving up use of functions means giving up overloaded and user-defined operators, so we make very little use of arrays of this type.

The other main type of allocatable array is the pointer array. The name probably strikes a certain amount of suspicious fear into pointer-wary Fortran programmers. However, Fortran90's use of pointers is in general simpler to understand and keep track of than that of C or C++. Fortran pointers are referenced and used in exactly the same way as the objects they point to. No confusion.

Either way, allocatable arrays are declared with the `dimension` attribute in addition to the usual `real`, `integer`, etc. Allocatable arrays also get the `allocatable` attribute, pointer arrays the `pointer` attribute. Conveniently enough, the actual allocation syntax is just like that understood by SEP's fortran preprocessors SAW and SAT.

```
integer :: n1,n2,n3
real, allocatable, dimension(:,:,:):: alloc_array  !3D allocatable array
real, pointer, dimension(:,:)    :: p_array !2D pointer array

allocate(p_array(n1,n2))
allocate(alloc_array(n1,n2,n3))
```

Structures

Structures mean the same thing in Fortran90 as in C. They give the user the ability to incorporate data of different types into several variables. This has obvious benefits. For example, in Three Dimensional Filtering (TDF), subroutines are always passed arrays and the arrays' dimensions. A simple use of a structure would be to define one that contained the dimensions as well as the array. Structures can make code much easier to read, by greatly reducing the numbers of arguments being passed to subroutines and functions. This same reduction also makes it possible to turn many subroutine operations (convolution, for example) into binary functions, which may be called as operators. Well thought out, this can result in very easy coding and expository code. The structures that we use in our SEP data structure are described elsewhere in this paper, but a simple example is given in the Modules paragraph below.

Modules

Modules are the key unifying element among the new tools in Fortran90. They are also the thing that replaces the most confusing parts of Fortran77. In particular, common blocks, the scourge of old Fortran programs brought on by the need for global variables without passing excessive numbers of arguments, can be completely done away with. A `module` provides an *explicit interface* to any program or subprogram

that uses it, which is to say that whatever data structures and/or subroutines are placed in a module are universally available, and in such a way that usage can be checked by the compiler to prevent the easy corruption that is a risk of common blocks.

A Fortran90 `module` consists of two parts. The first, the declaration part, is for data structures and interfaces. As an example, the code fragment below contains a simple data structure consisting of an allocatable real-valued vector and an integer, which would presumably be used to hold the length of the vector. It also contains an tt interface, which are used to overload operators, functions, and subroutines. The `interface` is named `myminus`, which is the name that a calling program would use to access one of the two `module procedures` listed. These are subroutines or functions that are coded in the second part of the module. When compiling a main program which calls one of these two subroutines or functions, the compiler decides which of the two to use in the executable based on the arguments given to the subroutine. If the arguments in the calling program match the first `module procedure`, that is used in the executable, if the arguments match the second, then that is used. Or in other words, compile time rather than run time checking.

```
module mymodule
type myvector
real, dimension(:), pointer :: vector
integer                     :: length
end type myvector
interface myminus
module procedure myminusvector, myminusscalar
end interface
```

The second part of the module is the subprogram part, which is where subroutines and functions are put. The subprogram part of the module begins with the word `contains`. Below is continued the example `module`, with two subroutines corresponding to the two `module procedures` listed in the first part of the module. The first multiplies a vector by negative one, the second multiplies a real number by negative one.

```
contains

subroutine myminusvector(inputvector)
type (myvector):: inputvector
inputvector%vector = inputvector%vector * -1.
end subroutine myminusvector

subroutine myminusscalar(inputscalar)
real :: inputscalar
```

```
inputscalar = inputscalar * -1.
end subroutine myminusscalar

end module mymodule
```

It is the *explicit interface*, half created by the module, that allows the checking required to resolve which subroutine to call. The other half of the interface comes from the calling program, which must include the line `use mymodule`. Note that the potential for ugliness brought on by `use`-ing many modules from a given program is easily averted, because modules may `use` other modules, and thus be chained together. Our SEP Fortran90 library contains many modules, which are joined by `use` association in a single module called `sep_f90_mods`. Any program which simply `use`s this one module will have access to all the modules in the library.

Intrinsics

Of more superficial value than modules and structures, but beneficial for their clarifying effect on code, are a number of array intrinsic functions introduced in Fortran90. Things like transpose, matrix multiplication, dot product, etc. are predefined as functions. Of course, these are all easy to code and forget as short subroutines in any language, but they are beneficial in this form nonetheless, because they are (in general, but dependent on the compiler) coded in assembler and very efficient, and because they are very clear and generally lead to code with fewer subroutine calls, arguments passed, and so forth.

STRUCTURE

Our data structure was designed with the goal of mirroring SEPlib90. Seismic data exists primarily as traces, regularly sampled in time, and regularly or irregularly sampled along various gridding axes, which may be spatial, such as midpoint and shot location; or non-spatial, such as shot number. In SEPlib90, separate grid and header files are used to represent the regular (binned) and irregular properties, respectively, of a given data set.

Our Fortran90 structure is similar, using two levels of defined data type to achieve this representation. The basic units are the trace, `sep_tr` and the header `sep_hdr`. The `sep_tr` type consists of a logical flag declaring whether or not that particular trace exists, and one each of complex, real, and integer valued allocatable vectors. One of these will be allocated for an existent trace, dependent upon the type of the dataset. The `sep_hdr` type is analogous, consisting of a logical flag which states whether or not a given header exists, and allocatable real and integer arrays for storing header keys. These types are like the data and header records of a SEPlib90 data set.

Structure is given to these random traces and headers through organization into larger arrays, which have axes analogous to SEPlib90 grid axes. A given data set actually has two such arrays, with each element in the array a pointer either to a trace, or to the headers for a trace. For very sparse data sets, missing trace locations may all point to a shared zero trace for economy of memory. The arrays have the dimensionality and axis parameters of the SEPlib90 data set which it will contain. Along with the grid arrays, a data set must have the axis parameters, number and format of header keys, certain logical flags for book keeping, etc. Seven types are defined, for one- to seven-dimensional data sets, which should easily accommodate most needs for seismic processing (since a 3D pre-stack data set requires only five-dimensional space).

The structure is most easily described graphically. This is done in Figure 1. Figure 1 displays the shape and connectivity of a two dimensional data set, such as a common midpoint gather. The figure shows a data set with four traces, binned onto a grid with six grid cells. The two empty cells share the zero trace, `zero_tr`, and zero header, `zero_hdr`. In the example there are three header keys, described by the three elements of the array `keys`, each of which consists of a name, type, and format (the identifying traits in a SEPlib90 header format file), and an index referring to location inside each `sep_hdr` array. It is not shown, but each `sep_hdr` can consist of real and integer components. Also not shown are various logical flags.

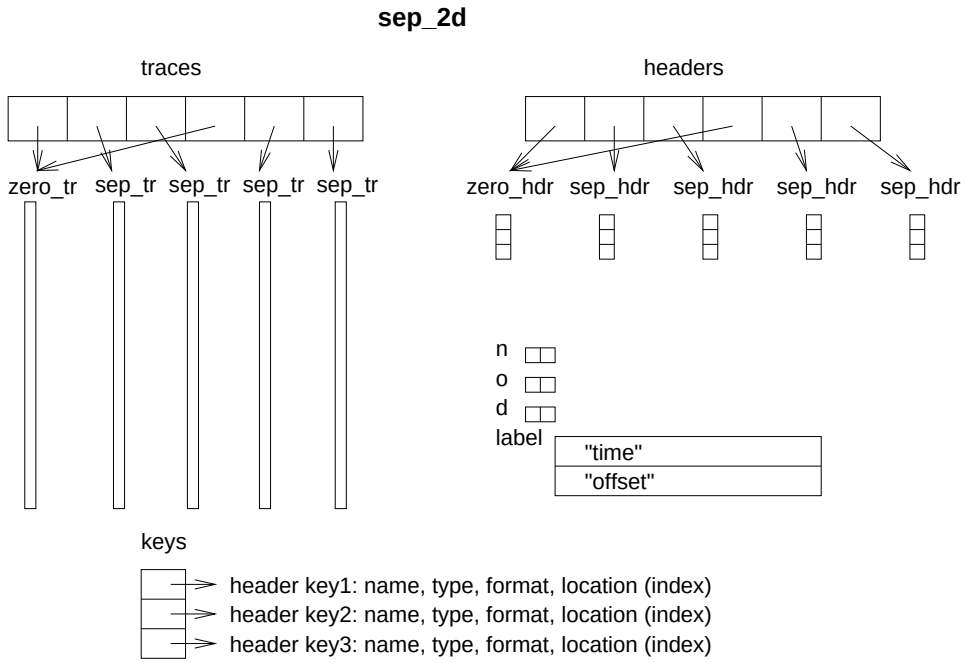

Figure 1: Diagrammatic representation of Fortran90 data structure for SEPlib90. bob1-struc2 [NR]

Below is the complete `sep_2d` structure:

```
type sep_2d
  type (sep_hdr), dimension(:), pointer   :: hdrs     !  1-D header array
  type (sep_tr),  dimension(:), pointer  :: traces    !  1-D trace array
  type (sep_tr)       :: zero_tr                      !  zero trace
  type (sep_hdr)         :: zero_hdr                  !  zero header
  integer, dimension(2)   :: n                        !  number of elements
  real,    dimension(2)   :: o                        !  first element location
  real,    dimension(2)   :: d                        !  sampling
  character(len=128), dimension(2):: label            !  axis description
  integer       :: n_keys                             !  number of keys
  integer       :: num_int                            !  number of integer keys
  type (sep_key_info), dimension(:), pointer :: keys  !  description of keys
  character(len=128)    :: type                       !  type of data
  logical :: alloc_hdr                                !  if headers allocated
  logical :: alloc_tr                                 !  if traces allocated
  logical :: alloc_keys                               !  if keys are allcoated
  logical :: rite_keys                                !  if keys info written
end type sep_2d


type sep_tr
  real,    pointer ,dimension(:)  :: rdata            !  real data storage
  complex, pointer ,dimension(:)  :: cdata            !  complex data storage
  integer, pointer,dimension(:)   :: idata            !  integer data storage
  logical       :: exists                             !  if trace is allocated
end type sep_tr

type sep_hdr
  logical       :: exists                           !  if headers allcoated
  real,    pointer, dimension(:)  :: rhdr            !  real header storage
  integer, pointer, dimension(:)  :: ihdr            !  integer header storage
end type sep_hdr

type sep_key_info
  integer       :: location                         ! location rhdr/ihdr array
  character(len=128)    :: name                     ! key name
  character(len=128)    :: type                     ! key type
end type sep_key_info
```

CONSTRUCTORS

The complexity of our structure, and our desire to maintain as much flexibility and simplicity in function calls possible, forced the construction of our sep-data types to be broken into multiple steps.

set_data_pars

The only routine that must be called is `set_data_pars`. The overlying grid and the type of data that will be contained within the grid is described by `set_data_pars`.

The grid is 'described' in traditional SEP format through the `n, o, d, and label` parameters. The `grid` can mean different things. In a traditional SEP77 data-set, with uniform spacing the meaning is quite clear. In Sep90 data-sets with a grid format file, our `sep_type` grid is analogous in shape and purpose to the grid described by the grid format file. With SEPlib90 files without a grid, and more generally any irregular datasets the grid describes any computational regularity that might be present in the dataset.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.
type	character(len=*)	Type of data contained in the structure.
n	integer (array of data dimensionality> 1	The extents of the data.
o	real (array of data dimensionality> 1	The origin of the data.
d	real (array of data dimensionality> 1	The sampling interval of the data.
label	character (len=*) (array of data dimensionality> 1	The axis labels for the data.

set_header_pars

SEPlib90 incorporated the concept of headers through the header format file and header value files. The keys are described by key name, type, format, and index in the header formal file. `set_header_pars` serves the same purpose as the header format file. It sets in the `key` structure the key names, types, and format and constructs a useful reference array that speeds up accessing of integer and real header values.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.
n_keys	integer	The number of header keys.
name	character(len=*) array(n_keys)	Array with key names.
format	character(len=*) array(n_keys)	Array with key formats.
type	character(len=*) array(n_keys)	Array with key types.

init_traces

`init_traces` allocates n-1 (where n is the dimensionality of the `sep_type`) trace array and sets the `rdata`, `cdata`, or `idata` to the corresponding member of the `zero_tr` (see Figure 2). This is one of the most powerful features of our formulation, the regularity of a grid, and the corresponding simplicity of accessing, is preserved, while:

- not allocating exorbitant memory and
- still allowing easy determination of holes in the grid.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.

init_header

Serves the same function as `init_header`. The header array is allocated and the `rhdr` and/or `idhr` portion of the `hdrs` are set to the corresponding `zero_hdr` value.

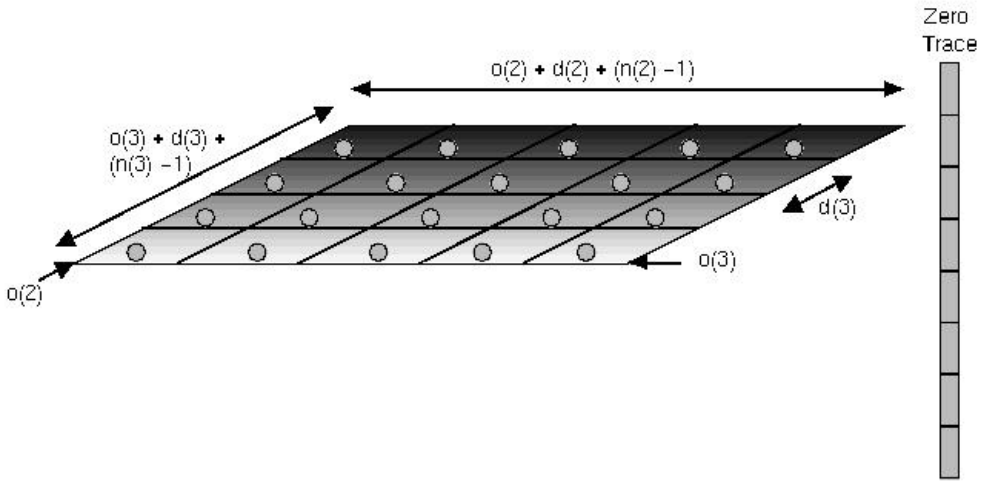

Figure 2: Diagrammatic representation of init_traces function, in this case a `sep_3d` structure. A 2-D array of the `sep_tr` function is allocated along with the zero trace. Each member of `sep_trace` is then set to the zero trace (gray circles correspond to grid cells where the trace is associated with the `zero_tr`). bob1-init [NR]

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.

allocate_traces

After calling `init_traces` all of the members of the trace structure are set to the zero trace. The `allocate_traces` routine allows allocating of portions, or the entire trace array structure (see Figure 3).

Figure 3: Diagrammatic representation of `allocate_trace`. Grey circles represent grid cells where the trace is still associated with the `zero_tr`.

The subroutine defaults to allocating the entire trace structure, but portions of the `traces` array can be allocated through several ways using optional parameters:

- use of standard SEP window arguments,
- or by passing a `tr_num` integer array, signaling holes with -1 values, the same way they are described in grid values file.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.
fwind	integer array	Optional, first location along given access to start allocating traces.
jwind	integer array	Optional, skip factor when allocating traces.
nwind	integer array	Optional, number of traces to allocate along each axis.
tr_num	integer array (dim=1)	Optional, array which indicates whether or not to allocate a given trace.

allocate_hdrs

Same functionality and arguments as `allocate_traces`.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.
fwind	integer array	Optional, first location along given access to start allocating headers.
jwind	integer array	Optional, skip factor when allocating headers.
nwind	integer array	Optional, number of headers to allocate along each axis.
tr_num	integer array (dim=1)	Optional, array which indicates whether or not to allocate a given headers.

DESTRUCTORS

For memory reasons, and to allow easy reuse of structures, a destroyer is also necessary. We built two destroyers, one for the headers and one for the traces.

cleanup_traces

`cleanup_traces` deallocates all allocated `rdata`, `idata`, `cdata` portions of the `traces` array. And then deallocates the `traces` array. After this call it is possible to reuse the same structure with same dimensionality or a different dimensionality.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.

cleanup_hdrs

Same functionality as `cleanup_traces`. All allocated headers are deallocated along with header array.

FUNCTION DESCRIPTION

Name	Format	Description
sepdata	sep_type	The name of the dataset.

ACCESSORS

Data

In building the accessor routines we were confronted with the opposing objectives of ease of use and efficiency. On the one hand you would ideally like to access

data, regardless of type, in a simple uniform manner. Such an approach would, however, require an "if" statement being evaluated with each access, something that is computationally wasteful, especially inside loops. Our decision was to offer the programmer either alternative. When speed is not an issue, a simple `data` call can be used to access the data. In most cases, the user will have for-knowledge of his data type, in which case we provide the simple routines: `rdata`, for real data; `cdata`, for complex data; and `idata`, for integer data.

- Description:

  Making data access simple and straight forward was one of our primary concerns in designing our Fortran90 environment.

- Specifications:

FUNCTION DESCRIPTION

Name	Type	Description
sepdata	sep_type	The name of dataset.
index1	integer	location within `sep_tr%hdrs` structure.
index2..7	integer	location with in `sep_tr` array.

- Examples:

  In order to make the transition for users simple, we built accessors to mirror array accessing in Fortran 77. In Fortran 77 you would access a 2-D array through:

  `array(i1,i2)`

  Using our Fortran90 library the general form of the call would be:

  `data(array,i1,i2)`

  Where array is a `sep_2d` type, and i1 and i2 are integers.

Header

- Description

  The introduction of multi-format headers to SEPlib is one of the key components of SEP90. It is also one of the most difficult parts of the SEP90 data structure to effectively incorporate into Fortran77. With our Fortran90 library, multiple data formats do not cause a problem. Our structure independently holds the two data formats currently supported by SEP90, integer and reals, and uses the key type structure to navigate the `rhdr` and `ihdr` routines.

- Specifications

FUNCTION DESCRIPTION

Name	Type	Description
sepdata	sep_type	The name of dataset.
key_num	integer	Key index or
key_name	character (len=*)	The key name.
index2..7	integer	Location with in `sep_hdr` array.

- Examples:

  To the user, accessing header routines is straight-forward. The user can either access header information through the key index or by the key name. To access the header in a 4-D data set the call would be simply.

  `header(sepdata,key_num,i2,i3,i4)`
  or

  `header(sepdata,"sx",i2,i3,i4)`

DATA I/O

One of the most attractive features of SEP77 was its straight forward I/O capabilities through `sreed` and `srite`. With the advent of SEP90 and irregular data capabilities

I/O suddenly became much more complex. Figure 4 shows an example of what needs to be done to read in a CMP gather from a SEP90 dataset. Even by following this procedure you will have lost the overall grid structure and are storing both real and integer header values in a common array, a far from ideal situation. With our structure the procedure is much simpler and powerful. Our `sep_reed` and `sep_rite` can be used to read in any type and portion of SEP dataset (data and/or header and/or grid). In addition the data can be read into any portion of one of SEP datasets. We take full advantage of Fortran90's optionals to make the reading as simple or as complex as necessary.

To read in a trace from a SEP77 dataset into `sep_1d` type, the call remains as straight forward as the traditional sreed:

```
sep_reed("in",data1d,"real")
```

With the additional benefits that the o,d, n, and label are also retained in the sep structure.

To read in the 2-D CMP gather shown earlier becomes just a slightly more complex exercise:

```
sep_reed("in",data2d, "real",fout=/pos1.pos2/)
```

A significant improvement over the C or Fortran 77 approach. In addition the overlying grid is preserved and the data and the headers can be accessed using a call very similar to the way uniform arrays are reference in Fortran 77.

sep_reed parameters

- **Required Parameter**

FUNCTION DESCRIPTION

Parameter	Type	Description
tag	character(len=*)	The tag of history file.
data	sep_type	Location to read in the data (real, complex,integer).
type	character(len=*)	The type of data to read in.

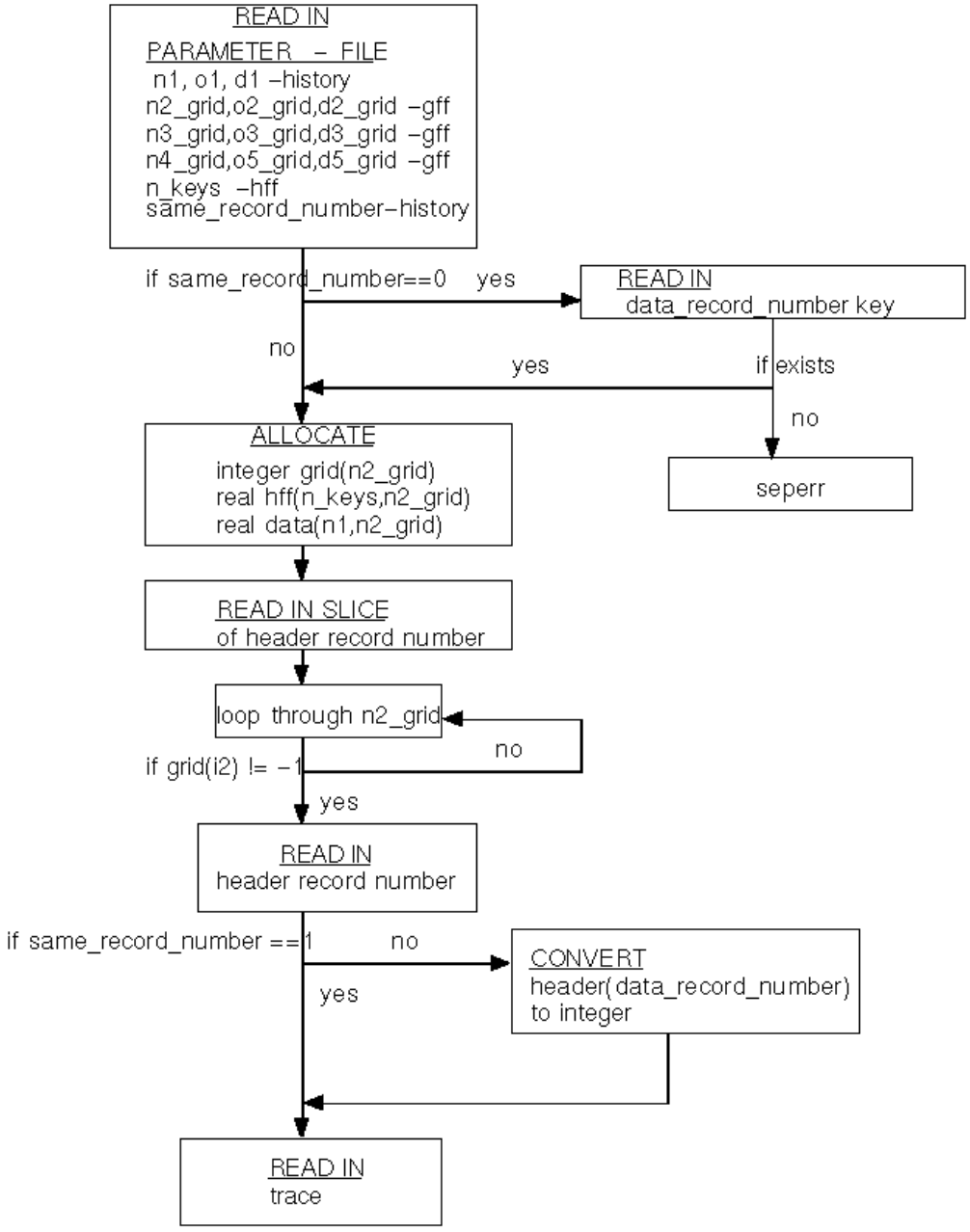

Figure 4: Flow chart of the procedure to read in a CMP gather. bob1-flow2 [NR]

- **Optional parameters**

FUNCTION DESCRIPTION

Parameter	Type	Default - Description
usegrid	logical	True if grid exists, whether or not to use the grid creating the data-set.
nin	integer array	Defaults to the values read in from the tag unless f and/or j defined.
fin	integer array	Portion of the dataset to begin reading in, defaults to 0.
jin	integer array	Sub-sampling of the dataset to begin reading, defaults to 1.
nout	integer (array if *data dim* > 1)	Size of *data*, defaults to amount of data read in.
fout	integer (array if *data dim* > 1)	Size of *data*, defaults to 0.
jout	integer (array if *data dim* > 1)	Size of *data*, defaults to 1.

- **Examples**

  To read in every other trace in the third slice from a 3-D real dataset the call would be:

  ```
  sep_reed(tag,data2d,fin=/0,0,2/, jin=/1,2,1/)
  ```

  To read in the first 50 traces of complex SEP90 dataset, ignoring the grid, padding to 1024 would be accomplished with:

  ```
  sep_reed(tag,data2d,nin=/800,50/,"complex",usegrid=.false.,
  nout=/1024,50/)
  ```

sep_rite

The structure of `sep_rite` is similar to that of `sep_reed`.

- **Required Parameters**

FUNCTION DESCRIPTION

Parameter	Type	Description
tag	character(len=*)	The ttag of history file.
data	sep_type	The location to read in the data.

- **Optional parameters**

FUNCTION DESCRIPTION

Parameter	Type	Default - Description
nin	integer (array if *data dim* > 1)	size of *data* to write out defaults to the entier data-set unless fin and/or jin are specified.
fin	integer (array if *data dim* > 1)	First index of *data* to write out, defaults to 0.
jin	integer (array if *data dim* > 1)	skip factor of *data* to write out, defaults to 1.
nout	integer array	Defaults according to fin,jin, and nin.
fout	integer array	Where to begin writing out.
jout	integer array	Skip factor when writing out.
writedata	logical	Whether or not to write trace info (defaults according to whether traces have been allocated).
writehead	logical	Whether or not to write trace info (defaults according to whether headers have been allocated).
writegrid	logical	Whether or not to write grid, default to true unless pad = true.
pad	logical	Whether or not to write out zero traces and/or headers. Defaults to .false.

- **Examples**

  An irregular dataset can be written out into a regular SEP77 type file, replacing holes with zeros by

  ```
  sep_rite(tag,data2d,pad=.true.,writehead=.false.)
  ```

  A 2-D slice from a *sep_3d* structure can be written out with:

  ```
  sep_rite(tag,data2d,fin=/0,0,3/)
  ```

OPERATORS

There were several directions that we felt we could follow with the design of operators. The first approach is what SEP traditionally did within Fortran77. Specifically to define all of operators through subroutine calls. This approach is advantageous because specifying whether to run the adjoint or the forward operator would involve changing one parameter in the subroutine call:

```
operator(input,output,adjoint)
```

On the other hand addition

is another valid operator and it not expressed through subroutines. We decided that it was more important to keep the operator concept consistent with its mathematical counterpart as much as possible.

This approach allows compound statements like:

```
D = A.conv.B - C
```

where A, B, C, and D are all `sep_type`'s.

Basic mathematical operators

So far we have used Fortran90s operator overload capacity to include the ability to add, subtract, multiply, and divide `sep_type`'s.

Operator	Usage
Addition	$C = A + B$
Subtraction	$C = A - B$
Multiply	$C = A * B$
Divide	$C = A/B$

Each operation checks that the dimensionality of the two spaces conform, add performs the desired operation. The traces are looped over, if both input traces are associated with zero trace, an output is not allocated at the given location, otherwise the give operation is performed and the output is stored in the associated output location.

Convolution

In addition to the four basic basic mathematical operators above, we have begun adding several addition useful operators such as convolution. Figure 5 and the following code fragment shows how easily operators can be incorporated into Fortran90.

```
cata= aata.transconv.bata

call sep_rite("input1",aata)
call sep_rite("input2",bata)
call sep_rite("output",cata)
```

In this case `adata`, `bdata`, and `cdata` are all `sep_types`.

SOLVERS

At this point we have only implemented one solver, SEP's traditional workhorse, the conjugate gradient solver(Claerbout, 1994).

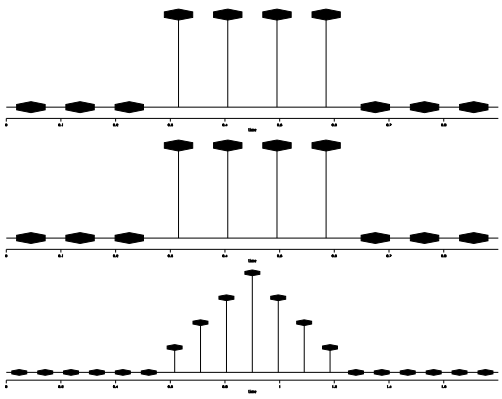

Figure 5: The two inputs and the output of a transient convolution. bob1-conv [CR]

Creating an effective interface for this subprogram presented a challenge because model space and data space may each be of any dimensionality, such as a model which is `sep_1d` and a residual which is a `sep_2d`; and both spaces are often made up of several parts, which may themselves be of different dimensionality. A problem may have, for instance, a data residual which is a vector and a model residual which is a plane. This presents far too many possibilities to consider an overloaded solver. Such a thing is at any rate needless, since the conjugate gradient algorithm needs only to take dot products; the Fortran77 incarnation of the solver assumes that whatever matrices it is given are vectors.

To handle diversely-dimensioned, compound spaces, we decided on a similar tactic. Anything being passed to `cg` should be in the form of a 2D data set, a vector of pointers to traces. This is accomplished by collapsing larger dimensionality data sets via the function `vectorize`. Several distinct spaces (a model residual and a data residual) may be `vectorized` and then joined together via the function `merge`. A call to subroutine `cg` would then look like this:

```
call cg(iter, niter, merge(vectorize(model),vectorize(filter)), &
merge(vectorize(dmodel),vectorize(dfilter)), &
merge(vectorize(modelresidual),vectorize(dataresidual)), &
merge(vectorize(dmodelresidual),vectorize(ddataresidual)))
```

In fact, a linear inverse problem could be entirely contained within the call to the solver, since procedures may be passed as arguments.

FUTURE

SEP is investing considerable effort in SEPlib90 in order to be able to do effective research into 3-D seismic problems. We think it is a good design and are committed to it. One of the early discoveries made as SEPlib90 was born was that "overhead"

coding had grown enormously. A great deal more programming, much of it predictable from program to program, was required to deal with header axes, grid axes, and the like. Fortran's lack of user-defined structures and operators engendered this as well as the need for declaring, passing, and keeping track of many new variables to deal with headers and grids. SEP knew fairly early that some other language was necessary, and we in particular chose Fortran90 for its familiarity, versatility, and its promised power – we had just taken delivery of a 16 node SGI power challenge, a platform where much attention has been paid to Fortran90 and its use as a parallel language.

The framework is in place now for use of Fortran90 at SEPlib90, though there remain numerous kinks. For the future we plan further development; a full set of operators, more routines for easily constructing the spaces needed by filters and residuals as necessary for inverse problems, and whatever else our experiences tell us we need.

CONCLUSION

Fortran90 is a powerful, parallelizable, object oriented language that lends itself easily into to SEP's vision for future 3-D processing, SEPlib90. The foundation that we have built takes advantage of the power of SEPlib90 in dealing with irregular datasets greatly simplifying the reading, writing, and accessing of the data while still intelligently managing memory. In addition our initial attempts at building operators using this data structure have proved to be simple, straight forward, and effective.

REFERENCES

Biondi, B., Clapp, R., and Crawley, S., 1996, Seplib90: Seplib for 3-D prestack data: SEP–**92**, 343–364.

Claerbout, J. F., 1994, Applications of Three-Dimensional Filtering: Stanford Exploration Project.

Nichols, D., Urdaneta, H., Oh, H. I., Claerbout, J., Laane, L., Karrenbach, M., and Schwab, M., 1993, Programming geophysics in C++: SEP–**79**, 313–471.

Schwab, M., 1994, Birth of a C++ project: SEP–**82**, 251–256.