Stanford Exploration Project, Report 93, November 11, 1997, pages 293–304

Stanford Exploration Project, Report 93, November 11, 1997, pages 293–304

SEPF90

Robert G. Clapp and Sean Crawley[1]

keywords: *interpolation, arrays, three-dimensional*

ABSTRACT

SEPlib90 gives programmers a storage format for irregular 3-D seismic data, and a library of accessor routines for bringing them into memory. However, experience showed early and often that I/O and data handling were cumbersome for SEPlib90 application programmers. Here we present a Fortran 90 library and data structure that overlay SEPlib90. The library simplifies various non-geophysical tasks such as data input and output; and makes windowing, allocation and deallocation of irregular traces, straightforward. In addition, the library serves as the link between SEP's new C++ framework, GOON (Geophysical Object-Oriented Numerics), and SEPlib90.

INTRODUCTION

Over the last few years SEP has been moving gradually into 3-D seismic processing. As a result the last year has seen a significant push at SEP in the development of SEPlib90(Biondi et al., 1996) to handle the irregular spatial sampling of 3-D data, and the increased use of Fortran 90, which is well suited for parallel computing, necessitated by the size and complexity of 3-D problems. The flexibility offered by SEPlib90 (multiple language and possible permutations) forces an extremely verbose coding style. In Clapp and Crawley(1996) we discussed our initial efforts in writing more convenient Fortran90 interfaces for interacting with irregular data. In this paper we present our revised work, referred to as SEPF90, which extends our initial efforts and moves towards a more object oriented approach. We present examples where SEPF90 is used in Fortran 90 main programs. In addition, we present examples where SEPF90 is used to write operators which can be called from C++ main programs under the GOON(Urdaneta and Karrenbach, 1996) framework, using the GOON class IGF90 (Irregular Gridded Function SEPlib90/Fortran 90). This combines the benefits of the conceptual power of C++ and the ease and computational power of Fortran 90.

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

The paper is divided in three sections. The first section covers the basic data structure. The second section covers the basic interfaces (both I/O and accessing routines). The third section presents an example of an IGF90 opperator and a main program which utilitze the SEPF90 library.

SEPF90

Terminology

Before describing the `sep_90` structure and routines it is helpful to define some terms:

- Operator overloading - A language feature that allows different routines, with different argument lists, to be called by the same name (like C++).

- Optional arguments - Arguments that may or may not be passed to a subroutine or function. When passed are usually specified by `my_optional = my_variable`.

- SEPlib77 dataset - traditional SEP old-style dataset which is described by

  - `n1, n2, n3...- integer` - the dimensions of the dataset.
  - `o1, o2, o3,..- real` - the origin of the axes of the dataset.
  - `d1, d2, d3,..- real` - the sampling between points on the axes.
  - `label1, label2, label3,...- character*` - label for dataset axes.

  and contains an array of (n1*n2*...) values.

- SEPlib90 dataset - extension of traditional SEPlib77 dataset to handle irregular dataset(Biondi et al., 1996). Consists of

  - a grid and grid format file (GFF), describing dataset regularities;
  - a header format file (HFF) and header values file, for header values;
  - and data values file and history file.

- `sep_90` - a Fortran90 type (similar to a C structure), which is the base type for accessing our library.

- SEPF90 - the SEP90 data structure and other contents of `libsepf90.a`.

STRUCTURE

The SEP90 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. SEP90's structure is similar to SEPlib90, using two levels of defined type to achieve this representation. The basic units are the trace, `sep_tr` and the header `sep_hdr`. The `sep_tr` is an allocatable one dimensional array. [2] One of these is allocated for each trace. The `sep_hdr` type is similar, consisting of pointers to real and integer allocatable 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. There are allocatable one dimensional arrays of type `sep_hdr` and `sep_tr`, labeled `traces` and `headers`, respectively. That is, `traces` is a one dimensional array, and each element of `traces` is a `sep_tr`. The one dimensional `traces` and `headers` are associated with an N-D grid through the allocatable arrays `n`, `o`, `d`, and `label`. These may reflect the history file, HFF, or GFF depending on the nature of the dataset. For economy of memory, missing traces are allocated to length 0. This enables safe array operations (no segmentation faults) without the expense of storing zeroes. Along with the grid arrays and axis parameters, a data set has space for number, names and formats of header keys, and a logical flag indicating whether or not a data set is sparsely sampled (if any traces have zero length).

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. 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.

Below is the complete `sep_90` structure. The brackets used to close each type are not part of standard Fortran 90, but are translated by ratfor90.

```
module sep_90_mod{
    type sep_tr {  #the type (structure) containing the data (seismic trace)
        real, pointer, dimension(:)   :: rdata #array containing the data
    }

    type sep_hdr {
```

[2] Actually a pointer array in proper Fortran 90 terminology, which is similar but not exactly equivalent to an allocatable array, but to avoid confusion in conversing with C programmers we write the more conceptually obvious term.

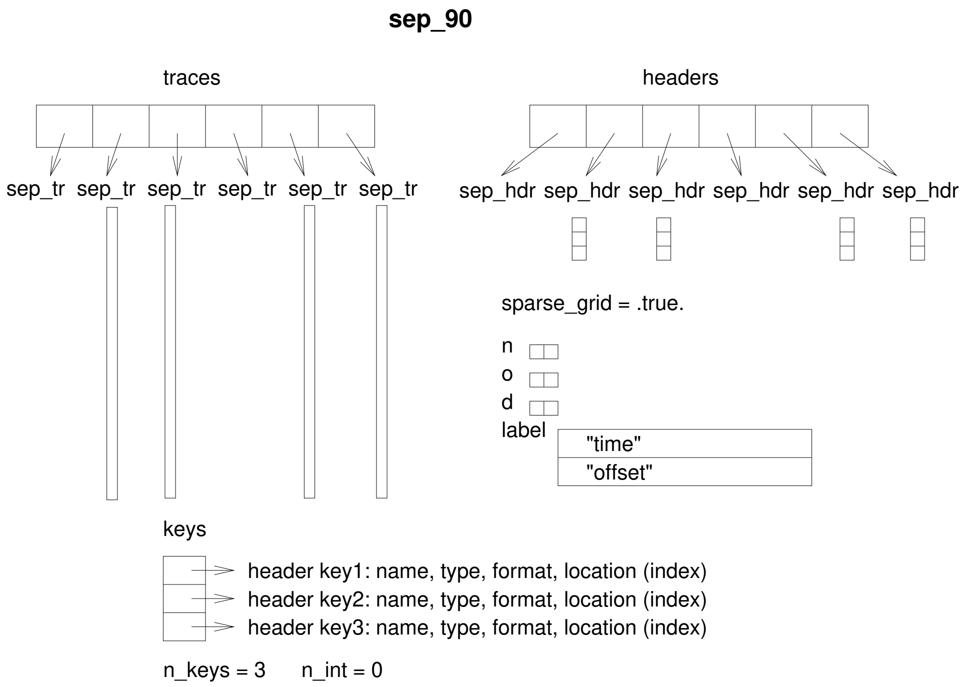

Figure 1: Diagrammatic representation of Fortran90 data structure for SEPlib90. bob1-s2 [NR]

```
    real,    dimension(:), pointer   :: rhdr  #containing real header values
                                              #associated with the traces
    integer, dimension(:), pointer   :: ihdr  #integer header values
}

type sep_key_info {                           #a structure containg
                                              #information about the keys
    integer              :: location          #location of the key inside
                                              #the rhdr and ihdr array
    character(len=128)   :: name,type,fmt
}

type sep_90 {
    type(sep_tr),  dimension(:), pointer   :: traces  #1-D array containg trace
                                                      #values
    type(sep_hdr), dimension(:), pointer   :: headers #headers associated with
                                                      #the traces
    integer, dimension(:), pointer         :: n       #length of data axes
    real,    dimension(:), pointer         :: o,d     #first sample and
                                                      #sampling interval
    character(len=128),dimension(:),pointer:: label   #description of axes
    integer        :: n_keys,n_int                    #number of keys associated
                                                      #with each traces and
                                                      #the number that are
                                                      #integers
    type(sep_key_info),dimension(:),pointer:: keys    #description of the keys
```

```
        logical :: sparse_grid                          #whether the grid has
                                                        #holes
    }
}
```

INTERFACE TO SEPF90

Like SEPlib itself the inner workings of the SEPF90 library do not need to be understood in detail. This section will describe the interface to the routines that the basic writer of operators or applications would use. A more complete description of the various routines can be found on the web.(Clapp and Crawley, 1996b)

Creating-Destroying

We provide several different levels of sophistication for creating `sep_90` types:

- `sep_create`

  `sep_create` provides the user the most flexibility in creating a `sep_90` type. It can be used to create

  - a simple data sub-space described entirely by its dimensions
  - a SEPlib77 simple dataset
  - a SEPlib90 irregular dataset with headers and traces
  - numerous mutations of the above (irregular without headers, only header information, etc.)

  `sep_create`, like most of the SEPF90 library takes full advantage of Fortran 90's overloading and optional abilities. As the user desires higher levels of complexity the call to `sep_create` expands:

  add `sepdata,n` To create a `sep_90` dataset where `o`, `d`, and `label` are extraneous. Where sepdata is a `type(sep_90), pointer` and n is an integer array describing its dimensions. To creating a dataset that is analogous to a SEPlib77 dataset.

  add `o,d,label.` To create a SEPlib90-like dataset

  add `n_keys, keyname, keytype, keyformat, mask (or tr_num)` . The user has to additionally supply information about the number (`n_keys`, integer), name (`keyname`, character array), type (`keytype`, character array), format (`keyformat`, character array), and potentially information about data irregularity by passing a logical or integer array describing where traces exist (`mask`, logical array or `tr_num`, integer array).

- sep_90_alloc and sep_90_duplicate

  These two routines will create a new sep_90 dataset from another sep_90 type. The first simply copies the shape, the second copies not only the shape but values. The interface to both takes the form:

  ```
  call sep_90_alloc(old,new)
  ```

- sep_reed

  The final, and easiest, method to create a sep_90 dataset is to simply read it off disk using the sep_reed routine described below.

All of these methods must be preceded a call to init_vars which allocates the sep_90(s) and sets up some initial defaults.

Freeing a sep_90 dataset is accomplished through a call to cleanup_sep

```
call cleanup_sep(sepdata)
```

I/O

One of the most complicated aspects in dealing with 3-D datasets in general, and SEPlib90 datasets in particular, is I/O. With a simple SEPlib77 dataset you could read in n, o, d, and label from the history file and have a complete description of the dataset. A simple call to sreed would then bring in all of the necessary data. With SEPlib90 datasets, its 6 files, numerous permutations, and its ability to deal with irregularity the task is much more daunting. What sep_reed provides is a way back to the "good old days". A simple call

```
sep_reed("in",sepdata)
```

will read in a SEPlib77 or SEPlib90, header or no headers, regular or irregular, traces in order or out of order dataset into the sep_90 type.

In addition with optional arguments the user can:

- read in a a subset of the dataset,
- to a subspace of a sep_90 type,
- read in only the headers,
- read only the 'space', or grid dimensionality,
- pad with zeroes in traces that correspond to SEPlib90 grid cell holes.

The entire list of options for sep_reed can be found on the web.

Just as sep_reed simplified reading in datasets, sep_rite and finish_tags making writing datasets easier.

sep_rite provides in its simplistic form a two argument call:

```
call sep_rite(tag,sepdata)
```

to write out any variety `sep_90` dataset. In addition it provides numerous optional arguments to write out subsets or too subsets of a SEPlib dataset on disk.

3-D datasets are usually irregular and are not possible to keep in core. As a result we generally process the dataset in sections, reading only what we need and writing what we are finished with. When writing these datasets to disk, the total number of traces and headers is not known until the very end. `finish_tags` provides the necessary mechanism to deal with this problem. Every time a call to `sep_rite` is made the number of traces written to a given tag is recorded. A call to `finish_tags` writes out to all the output tags the total number of traces written.

Accessing

In general the average user has no interest in the underlying storage mechanism and has no desire to reference through 3 levels of structures for both ease of use and speed reasons. As a result there is a class of routines for transferring information from the `sep_90` structure to and from simple integer and real arrays.

- `get_trace_vector` and `put_trace_vector`

  By specifying standard SEPLIB f, j, and n parameters will get and put a section of a `sep_90` dataset into a `real, dimension(:)` array.

- `get_trace_slice` and `put_trace_slice`

  Will retrieve or distribute a 2-D plane of data values.

- `get_trace_cube` and `put_trace_cube`

  Will retrieve or distribute a 3-D plane of data values (with the limitation that the dataset is regular or you wish to fill grid holes with zero values).

- `get_header_vector`, `put_header_vector`, `put_header_slice`, `get_header_slice`, `put_header_cube`, and `get_header_cube`

  Routines to put and grab header values into integer *and* real arrays.

- `get_key_values`

  Allows the user to read in a series of header values by passing either the key number or key name into an integer or real array.

Inspect

`Inspect` is a debugging tool that writes useful information about the dataset. `Inspect` has three different levels of operation denoted by the second argument:

```
call Inspect(sepdata,level)
```

- `level=0 Inspect` will give axis and key information.
- `level=1 Inspect` will check for data irregularity and print how many traces and headers exist.
- `level=2 Inspect` will print all trace and header values.

EXAMPLES

GOON style operators

Below is an example of a 2-D linear interpolation operator. The linear interpolation subroutine requires:

- position of the irregular values (sx and sy in this case)
- description of the mesh (n, o, d) that the values are being interpolated to.

In the old Fortran77-style operators, this information would be passed with every instance of applying the operator. In GOON a standard interface is necessary, as a result other information must be set at the initialization stage, and saved for use during operator execution. As a result GOON operators need to be written in modules:

```
module linear_interp2d_mod
  use sep_f90_lib_mod              !allows access to sepf90 library
  use sep_adjnull_mod              !sep90 style adjnull
!useful hold values
  real,save,private,pointer,dimension(:) :: dat_vec
  real,save,private,pointer,dimension(:,:) :: mod_vec
  real,save,private,pointer,dimension(:) :: f1,f2,g1,g2
  integer,save,private,pointer,dimension(:) :: ix,iy
  integer :: ntr
contains
```

To set up the operator we need to do some initialization, related to the C++ concept of instances.

```
subroutine interp2dinitn(model,data,key1_in,key2_in){
  character(len=*),intent(in)::key1_in,key2_in
  type(sep_90),pointer  ::data,model
  real,pointer,dimension(:) :: sx,sy
  integer,save,private         :: n2,n3
  real,save,private  :: o2,d2,o3,d3
  real ::tempr
  integer temp_int

!Try to find keys
  key1=find_key(model%keys,key1_in);key2=find_key(model%keys,key2_in)

  o2=model%o(2);d2=model%d(2) ;n2=model%n(2)  !set up some useful variables
  o3=model%o(3); d3=model%d(3);n3=model%n(3)  !for operator
  ntr=size(data%traces)

  nullify(mod_vec,dat_vec,sx,sy)  !nullify pointer arrays
  !Get sx and sy values
  call get_key_values(data,key1,sx); call get_key_values(data,key2,sy)

  !allocate arrays that describe irregular data's position on regular grid
allocate(f1(ntr),f2(ntr),g2(ntr),g1(ntr),ix(ntr),iy(ntr))
```

```
#set up
  do i2=1,ntr{
    tempr=sx(i2) - o2 /d2;temp_int=floor(tempr);ix(i2)= 1+temp_int
      if(ix(i2) > 0 && ix(i2) < n2) {
        f1(i2)= tempr-temp_int
        tempr=sy(i2) - o3)/d3  ;temp_int=floor(tempr) ;iy(i2)= 1+temp_int
        if(iy(i2) > 0 && iy(i2) < n3)  f2(i2)= tempr-temp_int;g2(i2)=1.- f2(i2)
else f2(i2)=0.  }
else f1(i2)=0.0
}
g1=1.-f1; g2=1.-f2
deallocate(sx,sy)
}
```

Finally there is the operator that actually performs the linear interpolation.

```
subroutine linear_interp2d(adj,model,data){ !their is no concept of add in
                                            !goon
  logical,intent(in) :: adj
  type(sep_90),pointer :: model,data
  integer :: i2
  call sep_adjnull(adj,model,data)  !call  a sep90 version of adjnull
  !transfer from a sep90 dataset into regular arrays
  call get_trace_slice(model,mod_vec);call get_trace_vector(data,dat_vec)
  do i2=1,ntr{
    if(.not. adj )
      dat_vec(i2)=dat_vec(i2) + g2(i2) *g1(i2) *mod_vec(ix(i2),iy(i2)) +
      f1(i2) * g2(i2) * mod_vec(ix(i2)+1,iy(i2)) + g2(i2) * f2(i2) *
      mod_vec(ix(i2),iy(i2)+1) + f1(i2) * f2(i2) * mod_vec(ix(i2)+1,iy(i2)+1)
    else{
      mod_vec(ix(i2),iy(i2))=mod_vec(ix(i2),iy(i2)) +
                              dat_vec(i2) * g2(i2) *g1(i2)
      mod_vec(ix(i2)+1,iy(i2))=mod_vec(ix(i2)+1,iy(i2)) +
                              dat_vec(i2) * g2(i2) *f1(i2)
      mod_vec(ix(i2),iy(i2)+1)=mod_vec(ix(i2),iy(i2)+1) +
                              dat_vec(i2) * f2(i2) *g1(i2)
      mod_vec(ix(i2)+1,iy(i2)+1)=mod_vec(ix(i2)+1,iy(i2)+1) +
                              dat_vec(i2) * f2(i2) *f1(i2)
    }
  }
!Put data back into septype
call put_trace_slice(model,mod_vec);call put_trace_vector(data,dat_vec)
}
```

The program `interp.r90` executes a single adjoint operation of the `linear_interp2d` operator above. Note the construction of the regular space, the initialization of the `sep_90` variables, and the application of the adjoint through the standard operator interface.

interp.r90

```
program testmain{
```

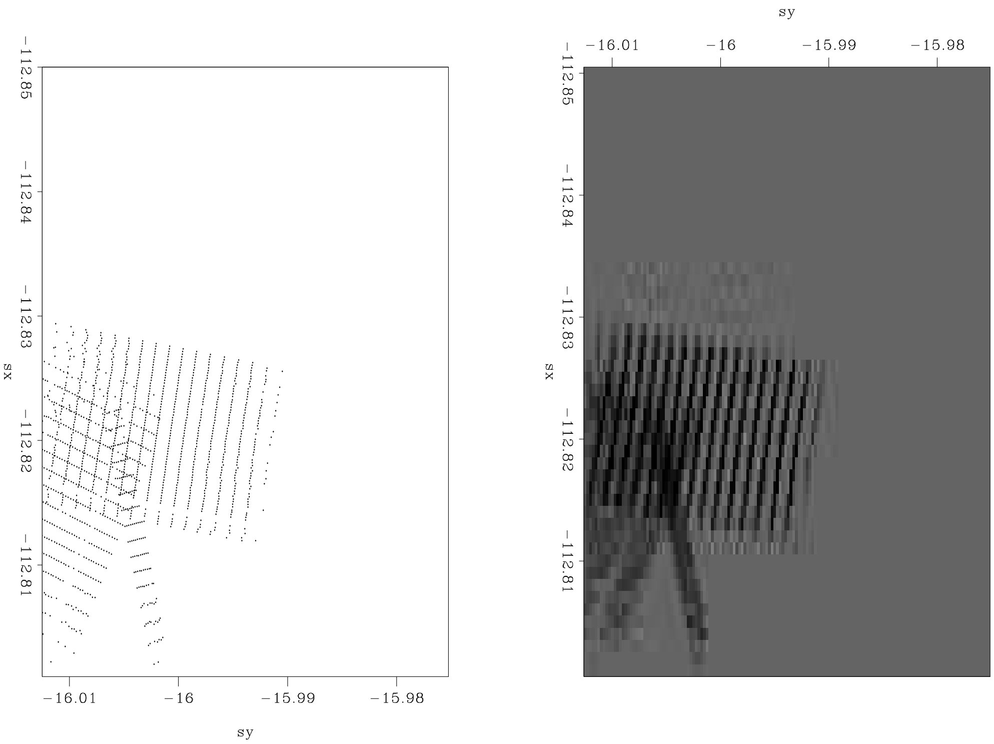

Figure 2: Left plot shows input sample locations, right plot shows the result of using the interpolation operator. bob1-interp2d [CR]

```
use sep_f90_lib_mod
use LINEAR_INTERP2D_MOD
implicit none
type(sep_90),pointer    :: d1,m1
integer,dimension(:),pointer            :: n
real,dimension(:),pointer               :: o,d
character(len=128),dimension(:),pointer :: label

call initpar()
call init_vars(d1,m1)  !initialize the SEP90 variables

!read in the data, ignore any gridding information
call sep_reed('in',d1,usegrid=.false.)

!--construct a model space
allocate(n(3),o(3),d(3),label(3))
n=(/1,50,150/)
o=(/0.,-112.849998,-16.012501/)
d=(/1.,.001,.000250/)
label(1)='trace';label(2)='sx';label(3)='sy'
call create_sep(m1,n,o,d,label,new_var=.true.)

!create the opperator
call init_linear_interp2d(m1,d1,'sx','sy')
!perform linear interpolation
```

```
call  linear_interp2d(.true.,m1,d1)
!write out the data
call sep_rite("out",m1)
!write out the = number of traces
call finish_tags()
}
```

Makefile example

To use the SEPF90 library there are some basic definitions that must be included in the makefile:

- `F90_INC_DIR = /wrksp1/seabeam/include/${MTYPE}`
  `UF90FLAGS = -I${F90_INC_DIR}`
  In order for the compiler to find the SEPF90 module files.
- `F90_LIB_DIR = /wrksp1/seabeam/lib/${MTYPE}`
  `UF90LIBS = -lsepf90 -l90auxf -l90aux`
  To link with the SEPF90 libraries.

FUTURE WORK/CONCLUSIONS

One can never say that a software project like this is finished, the only time it is not undergoing some change is when it reaches the stage of obsolence. With this condition stated, the SEPF90 library is fairly stable, it has been fairly well debugged, and most errors can now be contributed to the author of the main program and not to the library.

The only major addition/revision that we anticipate is the adding of the ability to handle virtual datasets. It is possible to extend the current library to handle a dataset that can not be kept in core, but instead is held mainly on disk, while being totally invisible to the user.

REFERENCES

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

Clapp, R. G., and Crawley, S., 1996a, Fortran90: Introduction and use in 3-D geophysical problems: SEP–**92**, 365–384.

Clapp, R. G., and Crawley, S., 1996b, Sepf90 documentation: http://sepwww.stanford.edu/sep/bob/f90.html.

Urdaneta, H., and Karrenbach, M., 1996, An IGF90 tutorial: SEP–**93**, 273–292.