Stanford Exploration Project, Report 92, November 12, 1997, pages 333–358

SEPlib90: SEPlib for 3-D prestack data

Biondo Biondi, Robert Clapp, and Sean Crawley[1]

keywords: *algorithm, three-dimensional, SEPlib, prestack*

ABSTRACT

We developed a generalization of the SEPlib software package capable to handle data with irregular spatial sampling, such as 3-D prestack seismic data. The new SEPlib, dubbed SEPlib90, is layered on the top of the old SEPlib, and shares with it the flexibility of use and the efficiency in accessing large amount of data. These attractive characteristics derive from two principles at the basis of SEPlib90 design: separation of the geometry information from the data traces, and exploitation of existing regularity, if any, in the data geometry.

INTRODUCTION

The SEPlib software package (Dellinger and Tálas, 1992; Claerbout, 1991) has been proven to be a very productive tool for seismic research and processing. However, its usefulness is fundamentally limited to processing regularly sampled data. This limitation is too restrictive when tackling problems in 3-D seismic and problems that involve geophysical data other than seismic. Therefore, we designed and implemented a generalization of SEPlib to make it capable of handling irregularly sampled data (from now on we will dub this new version SEPlib90, while the old version will be referred to as SEPlib77). Since the data geometry of a SEPlib77 data set is assumed to be regular, it can be described by few parameters such as axes lengths, origin coordinates and sampling rates. In SEPlib90, to describe the irregular data geometry, we associate each seismic trace with a trace header, as it is done in the SEGY data format, and in its many derivatives. However, to enable users and programmers to deal with irregularly sampled data with the same simplicity and efficiency that is characteristic of SEPlib77, SEPlib90 introduces the following two principles:

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

- Separate the geometry information from the seismic data.

  This simple but powerful idea is crucial for efficiently processing large amount of data, such as in 3-D prestack data sets. It allows us to minimize the access to the usually bulky seismic data files, while performing many useful operations on the trace headers and on specific subsets of the seismic traces.

- Exploit as much as possible the existing regularity in the data geometry.

  Regularity is important when analyzing and visualizing the data; further, it helps the development of simple and efficient code. SEPlib90 "regularizes" an irregularly sampled data sets by associating the data traces with a uniformly sampled multi-dimensional grid. This gridding information is then exploited by SEPlib90 application and utility programs to efficiently select and access the seismic traces.

Another important characteristic of SEPlib90 is that it is a generalization of SEPlib77 and not a completely new system. There are many good reasons for this choice. From the user point of view, it enables users familiar with SEPlib77 to quickly master SEPlib90. Further, it enables SEPlib90 to leverage the considerable amount of coding and brain power that went into SEPlib77. In particular, the layering of SEPlib90 over the SEPlib77 routines for accessing files (both ASCII and binary). SEPlib90 borrows some ideas from an early attempt of generalizing SEPlib77 to irregularly sampled data spearheaded by Martin Karrenbach (Nichols et al., 1994).

Clapp and Crawley present in this report (1996) an object-oriented Fortran90 package built on the top of SEPlib90. The coming to fruition of both these software projects promise to increase the efficiency of 3-D prestack seismic research at SEP, and possibly beyond.

In this paper we introduce the fundamental concepts underlying SEPlib90. We define the data format, describe the essential utilities, and the application program interface (API). To illustrate the concept we also present an example of using SEPlib90 for a trace filtering program and to sort and display a subset of a real 3-D marine survey.

DATA FORMAT

This section describes the data format of a SEPlib90 data set. A SEPlib90 data set is defined as the collection of all the files (ASCII and binary) that contain the information relevant to the geophysical data set.

Structure of a SEPlib90 data set

A "complete" SEPlib90 data set is made of 6 files, three ASCII files and three binary files. With the exception of the **History File**, the existence of all the other files is optional. The six files are connected to each other through pointers contained in the ASCII files. These pointers can be either simple UNIX file paths or more general URLs (the URL option is not implemented yet!). The path to the **Header Format File** (HFF) is specified by the value of the `hff` parameter in the History File. The path to the **Grid Format File** (GFF) is specified by the value of the `gff` parameter in the History File. Following is a brief "graphical" description of the connectivity among the 6 files, with the arrows $\Longrightarrow$ representing the ASCII pointers.

Header Format File (ASCII) $\Longrightarrow$ Header Values File (binary)

$\Uparrow$

$\boxed{\text{History File (ASCII)}}$ $\Longrightarrow$ Data Values File (binary)

$\Downarrow$

Grid Format File (ASCII) $\Longrightarrow$ Grid Values File (binary)

In addition to the links to the other files, the History File contains the processing history of the data set. The **Data Values File** (DVF) is defined as collection of fixed length records (**data records**) that contain the data values. Typically the data records are seismic traces.

The header values are stored in the **Header Values File** (HVF), that is defined as a collection of fixed length records (**header records**) describing the geometry and properties of the associated data records. The header parameters are described in the Header Format File by a table of **header keys**. A header keys specifies the name of the header parameter (**key name**), its data type (**key type**), and its position in the header record (**key index**). The association between the header records and the data records is described below.

If the data set has been binned on a regularly sampled grid, the **Grid Format File** contains the description of the grid. The **Grid Value File** contains the mapping information between the **grid cells** and the corresponding header records. The Grid Value File does not exist if the gridding is regular; that is, there is a one-to-one correspondence between grid cells and header records.

Data and Headers Coordinate Systems

The History File contains the usual SEPlib77 parameters $\mathtt{n}i$, $\mathtt{o}i$, $\mathtt{d}i$, $\mathtt{label}i$ (where i=[1,2,3,...]) describing the **Data Coordinate System**. The length of the axes in the Data Coordinate System must be constant and is given by the values of the respective $\mathtt{n}i$ parameter. The number of data values in a data record is given by `n1` and the the number of data records is equal to the product $(\mathtt{n2} * ... * \mathtt{n}i * ...)$.

The Header Format File also contains the usual SEPlib77 parameters $\mathtt{n}i$, $\mathtt{o}i$, `d`i, `label`i (where i=[1,2,3,...]) describing the **Header Coordinate System**. The number of header keys in the header records is given by `n1` and the the number of header records is equal to the product (`n2` $* \ldots *$ `n`i $* \ldots$).

Mapping between the header records and the data records

In general, the order and number of the data records stored in the Data Values File may be different than the order and number of the header records stored in the Header Values File. This happens, for example, if the Header Values File has been reshuffled (e.g. sorted or transposed) while the Data Values File was left untouched. Whether the data and header records are in the same order is indicated by the value of the integer parameter `same_record_number` in the History File. The value of `same_record_number` is equal to `1` if the records are in the same order, and equal to `0` if they are not. If `same_record_number` is missing from the History File it is defaulted to `1`. When the data and header records are in the same order (`same_record_number=1`), the association between header records and data records is given by the positions of the records in the respective binary files, and the Data Coordinate System coincides with the the Header Coordinate System. If data and header records are in different order the association between data records and header records is assured by the reserved header key `data_record_number`, that contains the **data record number** of the associated data record. The value of the data record number is defined as equal to the position of the data record in the Data Values File.

Gridding information

The Grid Format File and the associated Grid Values File define the **Grid Coordinate System** and they contain the information about coordinates of each header records in the Grid Coordinate System. The Grid Coordinate System is a regularly sampled coordinate system defined by the the parameters `n`i`_grid, o`i`_grid, d`i`_grid, label`i`_grid` (where i=[1,2,3,...]) in the conventional SEPlib style. The mapping between the grid cells and the header record can be either regular or irregular. A gridding is regular if for each grid cell in the Grid Coordinate System exists a header record, and vice versa, for each header record exists a grid cell. The grid cells and the header records are connected by tables of pointers to header records. These tables have an entry for each grid cell, containing the **header record number** of the corresponding header record. The value of the header record number is equal to the position of the corresponding header record in the Header Values File. Notice that if `same_record_number=0` the header record numbers are different from the data record numbers of the associated data record.

If the gridding is irregular, there are grid cells for which there is no associated header record. For these cells the pointer in the header record number tables are null (equal to `-1`). The following schematic illustrates the double-pointer mechanism that connects grid cells with data records, through the header records.

Grid Values File		Header Values File		Data Values File
-1 ↓				
header record number	↘			data record
-1 ↓		data record number	↗	
-1 ↓				
header record number	→	data record number	↘	
-1 ↓				data record

If the data is irregularly gridded the header record number tables are encoded in the Grid Values File. The format of the encoded tables is variable, and can be different from file to file. However, the programming interfaces for accessing the header record number are well defined and independent from the encoding. The encoding is variable because the optimal encoding strongly depends on the sparsity of the grid cells within the Grid Coordinate System, and thus depends on the particular way the header records were binned into the Grid Coordinate System. The gridding tables can be stored and retrieved by using the functions `sep_get_grid_window` and `sep_put_grid_window` described below.

The Grid Format File and the Grid Values File are optional. When no Grid Format File is associated with a data set, the Grid Coordinate System is assumed to be the same as the Header Coordinate System, and the grid coordinates of the header records are assumed to be regular.

Header Conventions

The header keys in SEPlib90 are the key to our solution to deal with irregular data. In many cases we will superimpose a grid to ease navigation through a data set (for example we might grid the offset access) , but the main routine will use the higher order of accuracy provided by the header values. SEPlib90 is designed to be as flexible as possible, the user is allowed to create as many header keys, of various types, in any order he wishes. This flexibility is one of the most attractive elements of SEPlib90, but complete anarchy is also unwanted. Without some standards a core set of programs will require an enormous set of user supplied parameters. For example, a NMO routine would require you to supply

- common midpoint key(s),
- offset key(s),
- data record number key,

- common midpoint axis/axes, and
- offset axis/axes.

Such a situation would detract too much from the ease of use that SEPlib is known for. As a result we have reserved certain key names that our core set of programs recognize. To allow ease in transferring from different data types we have based a number of our naming on Colorado School of Mines' Seismic Unix (SU) standards which in turn is based on (Barry et al., 1975).

Name	Type	Description
sx	integer	Source x location
gx	integer	Receiver group x location
sy	integer	Source y location
gy	integer	Receiver group y location
trid	integer	Trace status (1= exists 0=padded)
offset	integer	offset
cdp	integer	cdp ensemble number
cdpt	integer	trace number within ensemble

In addition, we have included several other header key conventions that make traversing the data easier, and/or find common use in 3-D processing:

Name	Type	Description
data_record_number	integer	Location of trace of given header record
s_x	float	Source x location
g_x	float	Receiver group x location
s_y	float	Source y location
g_y	float	Receiver group y location
cmp_x	float	Common mid-point x location
cmp_y	float	Common mid-point y location
azimuth	float	Source-receiver azimuth
offset_x	float	Offset in x location
offset_y	float	Offset in y location

UTILITIES

A set of basic utility programs have been developed together with the library. These programs can be used to perform useful operations on SEPlib90 data sets. The following is a partial list of these utilities, that are in different stages of development.

- `Synch90` - Reorder the data traces according to the value of the `data_record_number` in the trace headers.
- `Sort90` - Sort the headers and the traces in a data set, and create the gridding information.
- `Window90` - Window a data set according to the header values and/or the grid information.
- `Transp90` - Transpose the axes of the Data Coordinate System and/or of the Grid Coordinate System.
- `Infill90` - Regularize a data set by placing zero traces in the empty grid cells.
- `Headermath` - Perform arithmetic operations on header values.
- `In90` - Print information on a data set.
- `Cp90` - Copy a data set.
- `Rm90` - Remove a data set.
- `Create90` - Create a SEPlib90 data set from SEPlib77 files that contains the data traces and the geometry information
- `Su2sep90` - Convert a SU data set into a SEPlib90 data set.
- `Sep902su` - Convert a SEPlib90 data set into a SU data set.
- `Promax90` - Converts a Promax data set into a SEPlib90 data set.

ACCESSOR ROUTINES

This section describes the routines that can be used by the application programmer to access the information contained in a SEPlib90 data set: data, headers and grids.

Accessors to coordinate system parameters

These routines access the parameters that describe the data axes, the header axes and the grid axes contained in the History File, the Header Format File, and the Grid Format File (e.g. `n2_grid, o2_grid, d2_grid, label2_grid`).

Accessors to header formats

Routines that access the table of header keys stored in ASCII format in the Header Format File.

Accessors to header values

These routines access the header values either by key name or by key index. The haeder values can be accessed either one header key at the time, or a specified number of whole headers at the time. The pointers to the desired header records can be computed using the headers navigation routines.

Headers navigation routines

One of the main additional complexities caused by the introduction of irregularly sampled data is that navigating the data set, that is computing the pointers into the Data Values File and Header Values File of the records of interest, becomes a non-trivial problem. Since pointers to the data record are stored in the data record number header key of the corresponding header record, the problem is reduced to compute the pointers to the header record, that we call header record number. This task is facilitated if the data set has been gridded. In this case the programmer is provided with two simple headers navigation routines (`sep_get_grid_window and sep_put_grid_window`), for accessing the header record numbers given the corresponding grid coordinates.

The modality for storing and retrieving the information on the grid coordinates from the database is straightforward. The header record number are accessed for regular sub-windows of the grid coordinates. The sub-windows are specified by using the following windowing parameters for each axis of the Grid Coordinate System: `n_wind` (length of the window), `f_wind` (origin of the window) and `j_wind` (undersampling rate of the window).

Accessors to file links

These routines provide access to the links between the History File, the Header Format File, and the Grid Format File, that are specified as ASCII strings in the History File.

TRACE FILTERING EXAMPLE

As an example of SEPlib90 data access, below is presented a short program called `Filter` which performs smoothing via triangle filtering. Though the code is written in Fortran90, it uses almost exclusively Fortran77 functions, and but for a few lines could be compiled with `f77`. The only features unfamiliar to Fortran77 programmers should be the array declarations of variables `trace` and `grid`, the presence of an optional subroutine argument, and the use of a module.

Aside from variable declarations, there are really just two parts into this program. The first is a series of error checking statements, each of which calls the program-halting subroutine `seperr` if some condition is not met. SEPlib90-specific subroutines are identifiable in the code by their naming convention, using multi-word phrases beginning with "`sep`". The first of these, `sep_get_number_grid_axes`, returns a zero value if successful, and some non-zero integer value if it fails. The value returned can be used to provide some diagnosis of the problem if one exists, but in this case the program simply aborts any time that the value is non-zero, with the message "Trouble getting grid axes." Assuming that no problems occur, program `Filter` stores the number of grid axes (obtained by `sep_get_number_grid_axes` from the grid format file) in the variable `num_axes`. There is then a further check that aborts the program if `num_axes` falls outside the range (2,3,4,5,6). A quick glance at the main loop structure reveals why this is done.

Notice that nowhere are the familiar SEPlib `hetch` or `from history` statements that are used to get information on the input data's dimensionality. This job is instead carried out by the subroutines `sep_get_data_axis_par` and `sep_get_grid_axis_par`, which appear next in the code. The length and sampling of the time axis is determined by the former. Similar information for whatever spatial (or other) axes exist in the data are determined by the latter. This assumes that the data are regular according to the gridding coordinate system determined in the grid format file. This would be a foolhardy assumption for most processes, where spatial information is important and would be gotten from the headers. However, `Filter` works solely on the time axis, and is a nonphysical operation, such that precise information about source and receiver coordinates, etc. is unnecessary.

The final error checking is for the existence the header key, `data_record_number`. Because SEPlib90 header and data are stored separately, and may be manipulated independently, the possibility exists that the headers will be ordered differently on disk than the data (and there may even be fewer headers than traces). The header key `data_record_number` serves as a pointer from a given header to a trace. Program `Filter` checks for the existence of a header called `data_record_number`, and if it finds such a header key, it assumes that headers and data are not stored in the same order. To deal with this, the header key index of `data_record_number` is stored as the variable `data_key`, and the logical variable `in_order` is set to false.

The remainder of the program is fairly simple. Two arrays are allocated. One, named `trace` exists to temporarily store each input trace for filtering. The other, named `grid`, is an integer vector containing a subset of the header record numbers in the input data. Also, `nwind`, `jwind`, and `fwind`, parameters to the subroutine that gets header record numbers from the grid (`sep_get_grid_window`), are set to default values. Finally, each axis of the grid is looped over. Header record numbers are grabbed a vector at a time from the grid and stored in the array `grid`. The array grid is then looped over, each trace being input, filtered, and output.

```
program Filter
use filtmods
implicit none
real   ,dimension(:),pointer :: trace
integer,dimension(:),pointer :: grid
integer                       :: n(6)
real                         :: d(6),o(6)
integer                     :: i1,i2,i3,i4,i5,i6
integer                         :: sep_get_number_grid_axes
integer :: sep_get_grid_axis_par
integer                         :: getch,sep_get_data_axis_par,fetch
integer                         :: sep_get_key_index,putch,esize
integer                         :: sep_get_grid_window
integer                         :: same_record_number,num_axes
integer                         :: length,data_key
integer                         :: fwind(6),jwind(6),nwind(6),ierr
logical                         :: in_order
character(len=128)              :: label(6)

call initpar()

if(0==fetch("esize","d",esize)) then
        esize=4
end if

if(0.ne.putch("esize","d",esize)) then
        call seperr("trouble writing to output history file")
end if

if(esize .ne. 4) then
        call seperr("this program can only deal with real data")
end if

if(0 .ne. sep_get_number_grid_axes("in",num_axes)) then
        call seperr("Trouble getting grid axes")
end if

if(num_axes >6 .or. num_axes <2  ) then
        call seperr("Not equipped to deal with given number of axes")
end if

n=1
```

```
if(0 .ne. sep_get_data_axis_par("in",1,n(1),o(1),d(1),label(1))) then
        call seperr("Trouble getting data axis information")
end if

do i1=2,num_axes
        if(0 .ne. sep_get_grid_axis_par("in",i1,n(i1),o(i1),d(i1), &
label(i1))) then
                call seperr("Trouble getting grid axis information")
end if
end do

if(0==getch("length","d",length)) then
        length=4
end if

if(0==fetch("same_record_number","d",same_record_number)) then
        same_record_number=1
end if

if(same_record_number == 0) then
        in_order=.false.
        ierr=sep_get_key_index("in","data_record_number",data_key)
        if(ierr .ne. 0) then
                call seperr("Data is out of order and cannot find  &
data record number key")
        end if
else
        in_order=.true.
end if

allocate(grid(1:n(2)))
allocate(trace(1:n(1)))

jwind=1
nwind(2:num_axes-1)=1
fwind(1) = 0
nwind(1)=n(2)

do i6=1,n(6)
 fwind(5) = i6 -1
 do i5=1,n(5)
  fwind(4) = i5 -1
  do i4=1,n(4)
   fwind(3) = i4 -1
   do i3=1,n(3)
```

```
    fwind(2) = i3 -1
    call sep_get_grid_window("in",num_axes,n(2),nwind,fwind,jwind,grid)
           do i2=1,n(2)
           if(grid(i2) .ne. -1 ) then
                  call get_trace("in",in_order,n(1),esize,grid(i2), &
                                                trace,data_key=data_key)
                  call filter_trace(trace,n(1),length)
                  call put_trace("out",n(1),esize,grid(i2),trace)
           end if
     end do
   end do
  end do
 end do
end do

!if(0 .ne. putch("out",same_record_number,1)) then
if(0 .ne. putch("same_record_number",'d',1)) then
same_record_number=1
end if

end program Filter
```

Four subroutines are called within the inner loop. Subroutine `filter_trace` is simply convolution and will be skipped over here. Subroutine `get_trace` is a short routine which checks to see if the data and headers are stored in the same order, and acts accordingly to call `sseek` and `sreed`. Subroutine `put_trace` handles the output in a straightforward manner. Notice that even in the case that the input headers and data appear in different orders, the output will be in the same order.

Finally, subroutine `sep_get_grid_window` is the least obvious in its usage. It is complicated by the fact that the one axis (time axis) is implicit in a SEPlib90 grid. A data set gridded according to source x and source y header values only is actually described as having three grid axes. A look into the grid format file would show that source x and source y were the two and three axes, respectively. The one axis is not described in the grid format file. What this means is that the `num_axes` argument to the `sep_get_grid_window` subroutine is an integer whose value is one greater than the length of the vectors `nwind`, `jwind`, `fwind`, and `n`.

```
module filtmods
implicit none

contains

subroutine get_trace(tag,in_order,n,esize,record_number,trace,data_key)
```

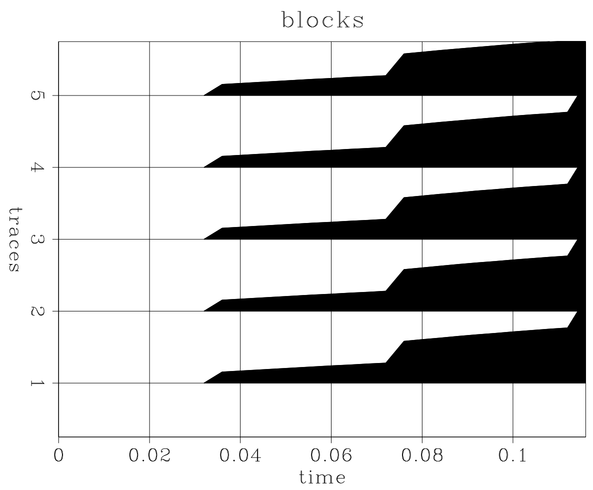

Figure 1: Input to the program Filter.x biondo2-blocks [CR]

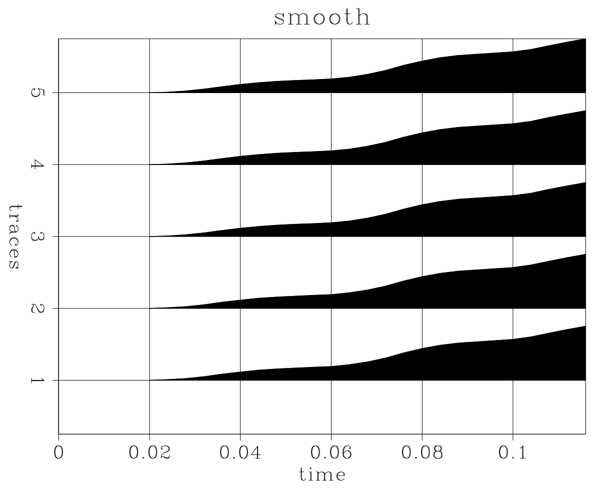

Figure 2: Output of the program Filter.x biondo2-temp [CR]

```
implicit none
character(len=*)          :: tag
logical                   :: in_order
integer                   :: n
integer                   :: esize
real                      :: trace(n)
integer, optional         :: data_key
integer                   :: record_number,sep_get_val_by_index
integer                   :: data_record_number,ierr
integer                   :: sreed,sseek
integer*8                 :: sseek_pos
real                      :: tempa(512)

if(.not. in_order) then
        if (.not. present(data_key)) then
                call seperr("must supply a data key if the traces &
                                              are out of order")
                end if
        end if

if(in_order) then
        sseek_pos = (record_number -1) * esize * n
else
        if(0 .ne. sep_get_val_by_index(tag,record_number,data_key, &
1, data_record_number)) then
                call seperr("trouble getting data_record_number ")
                end if
        sseek_pos = (data_record_number -1) * esize * n
        end if

ierr=sseek(tag,sseek_pos,0)
ierr=sreed(tag,trace,n*esize)

end subroutine get_trace

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

subroutine put_trace(tag,n,esize,record_number,trace)
character(len=*)          :: tag
integer                   :: n
integer                   :: esize
integer                   :: record_number,sep_get_val_by_index
real                      :: trace(n)
integer                   :: data_record_number
```

```
integer*8                 :: sseek_pos

sseek_pos = (record_number -1) * esize * n
call sseek(tag,sseek_pos,0)
call srite(tag,trace,n*esize)

end subroutine put_trace

end module filtmods
```

3-D PRESTACK EXAMPLE

The following example shows how to use SEPlib90 programs and libraries to analyze and manipulate 3-D prestack seismic data sets.

The data set is a single sail line belonging to a 3-D marine data set recorded in the North Sea. The data were recorded by shooting from two sources located at the stern of the recording ship into three streamers being pulled behind the ship. There are about 120,000 traces with 500 time samples in the data. The total size of the Data Values File is about 250 Mbytes. Figure 3 shows the source locations, for one trace out of ten in the data set. Figure 4 shows the receiver location for the same subset of traces in Figure 3. The receiver locations are scattered along the cross-line direction (y) because of a fairly strong cable feather, probably caused by ocean currents during acquisition. The corresponding CMP locations are shown in Figure 5. The cable feather caused a noticeable irregularity in the CMP coverage, with cross-lines overlapping in many places. Figure 6 shows the trace offset-azimuth distribution. Each of the six different trends distinguishable at small offsets corresponds to a source-streamer pair.

Exploring the data

At the start the data set is in SEPlib90 ungridded format. The seismic traces and the trace headers are in arbitrary order. Our goal is to analyze the information on the geometry of the data and to extract meaningful subsets of the data to be displayed. However, since the data set is relatively bulky (250 Mbytes), we want to explore it without replicating the seismic data themselves; further we want to avoid to perform the sorting and binning operation every time we choose to display a different subset.

The first step is to sort and bin the trace headers on a regular grid, by using the `Sort90` program. We define a three-dimensional grid, with the grid axes being the absolute value of the offset, the in-line midpoint coordinate, and the cross-line midpoint coordinate. Since for 3-D data the source-receiver offset is a vector, we could define azimuth of this vector as an additional grid axis; we leave this possibility as an exercise to the reader. The midpoint distribution is irregular (Figure 5), therefore

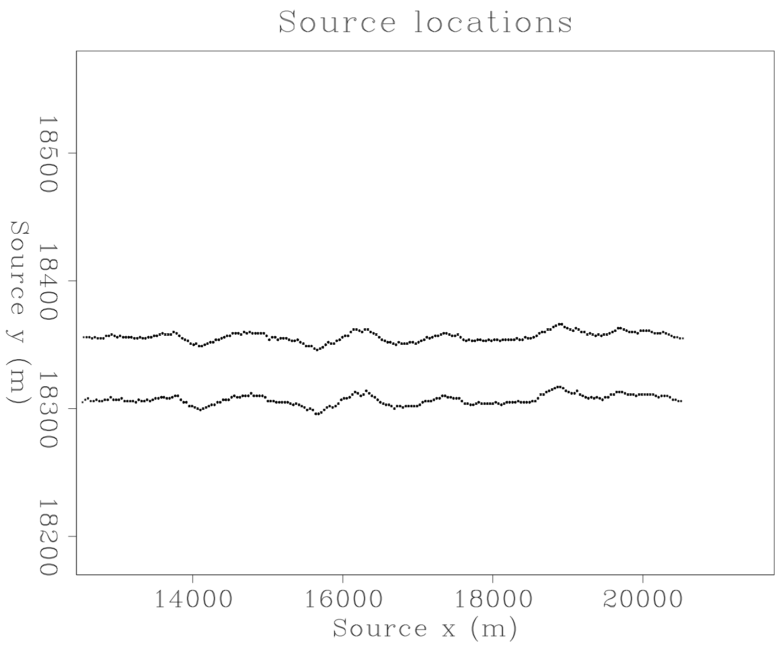

Figure 3: Source locations of data traces biondo2-Sxy [CR]

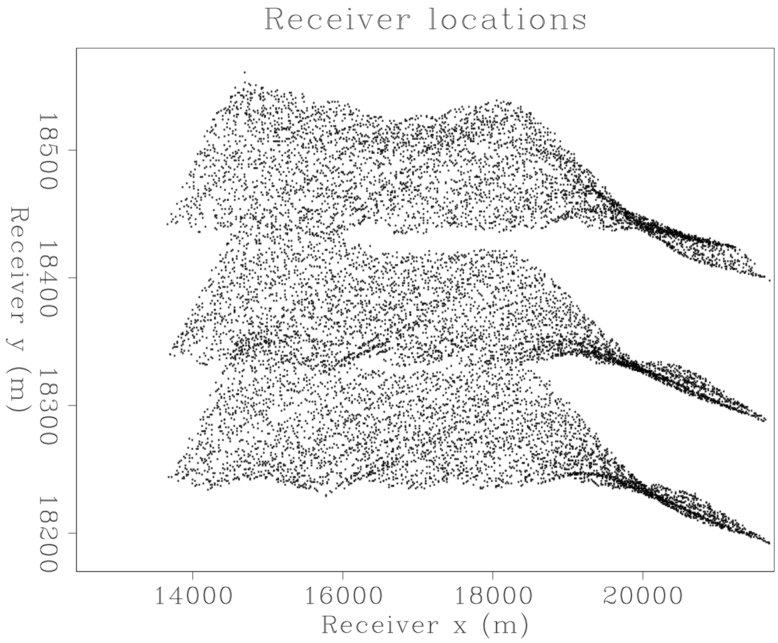

Figure 4: Receiver locations of data traces biondo2-Rxy [CR]

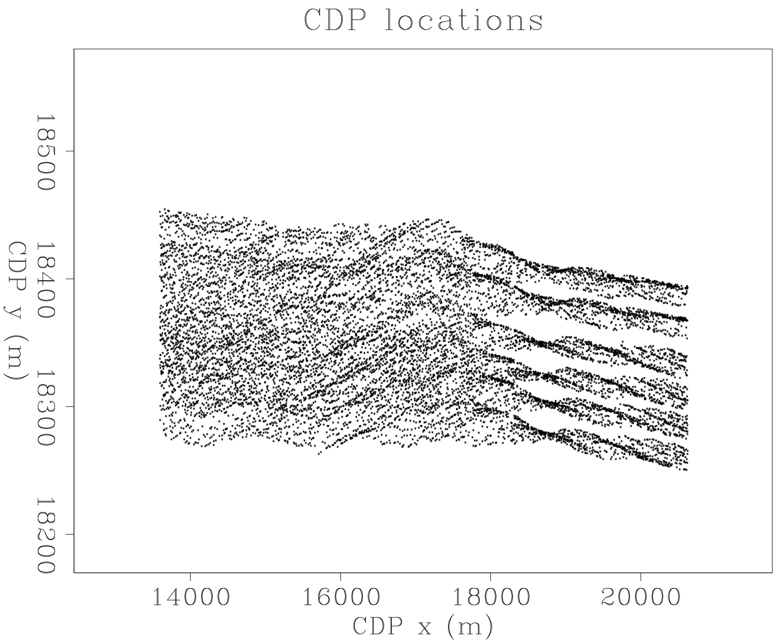

Figure 5: CDP locations of data traces biondo2-CDPxy [CR]

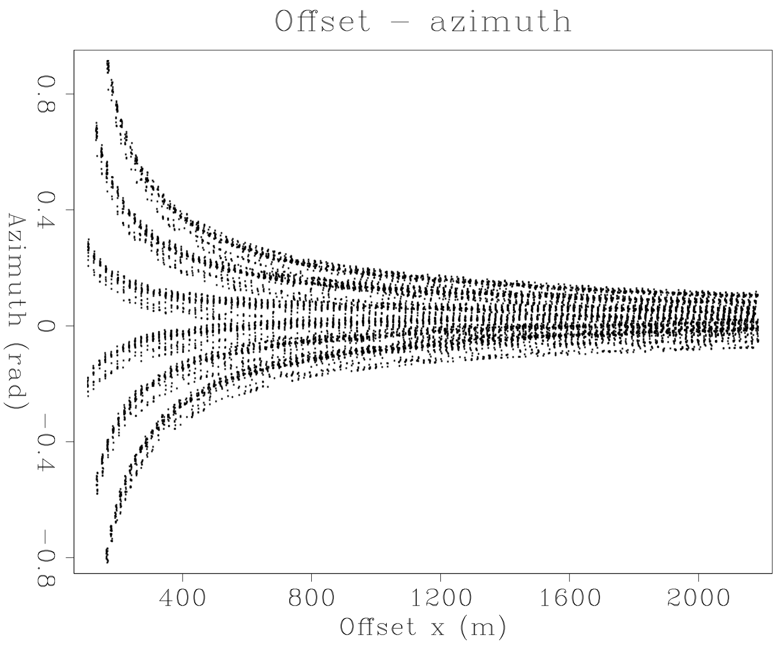

Figure 6: Offset-azimuth of data traces biondo2-OFFAZ [CR]

it is likely that more than one trace belong to the same grid cell. If this overlap happens, `Sort90` automatically adds an axis (called `trace_number_in_bin`) to the grid, making the grid a four-dimensional object. Notice that the sorting and binning are performed on the trace headers and do not require access to the seismic traces. However, as an option, `Sort90` can reorder the data themselves after having sorted and binned the headers.

Once the grid has been computed we can extract many interesting subsets of the data by using the windowing program `Window90`. This program can extract specified slices from the four-dimensional grid created by `Sort90`. For example we can display the subset of traces that correspond to grid cells with the same offset and cross-line midpoint. Figure 7 shows one of this "common-offset" sections. We can see the complex structures below the surface, with plenty of faultings and diffractions. Similarly we can extract a CMP gather at any desired locations, as the one shown in Figure 8. Because the sorting and the binning have been done once by `Sort90`, and the pointers to the data traces corresponding to the grid cells are stored in the Grid Value File, `Window90` can extract these subsets very quickly without the need to go through the whole data set, but just by extracting the desired traces from the Data Value File.

Analyzing binning parameters

The choice of the binning parameters for a data set depends on many factors, such as the acquisition geometry and the planned subsequent processing flow. Using `Sort90` we can study the effects of choosing different binning parameters in order to select optimal ones. For example, we can vary the bin width along the midpoint axes to analyze the the trade-off between bin resolution and uniformity in the offset and azimuth coverage within the bins. `Sort90` is an efficient tool for this study, because it bins the trace headers and creates the gridding information, without accessing the bulky Data Values File. To assess the quality of a set of binning parameters we can display subsets of the data, as described in the previous section. We can also examine the distribution of the trace headers on the grid by measuring the fold within each of the grid cells. A simple way of measuring the fold is to count the number of elements along the `trace_number_in_bin` axes that `Sort90` creates when encountering multiple traces belonging to a single grid cell. The following subroutine performs this task, and outputs the results into a SEPlib77 data set, which can then be displayed and examined. Notice that the subroutine is coded in Fortran 90, and uses a few of the powerful features of the language (e.g. automatic allocation of arrays, reduction intrinsics) that enables the writing of simple and compact code.

```
! subroutine that computes the fold from the gridding information
subroutine comp_fold(num_grid_axes,tot_grid_cells,n_in_bin,n_grid)

implicit none

integer sep_get_grid_window,srite
```

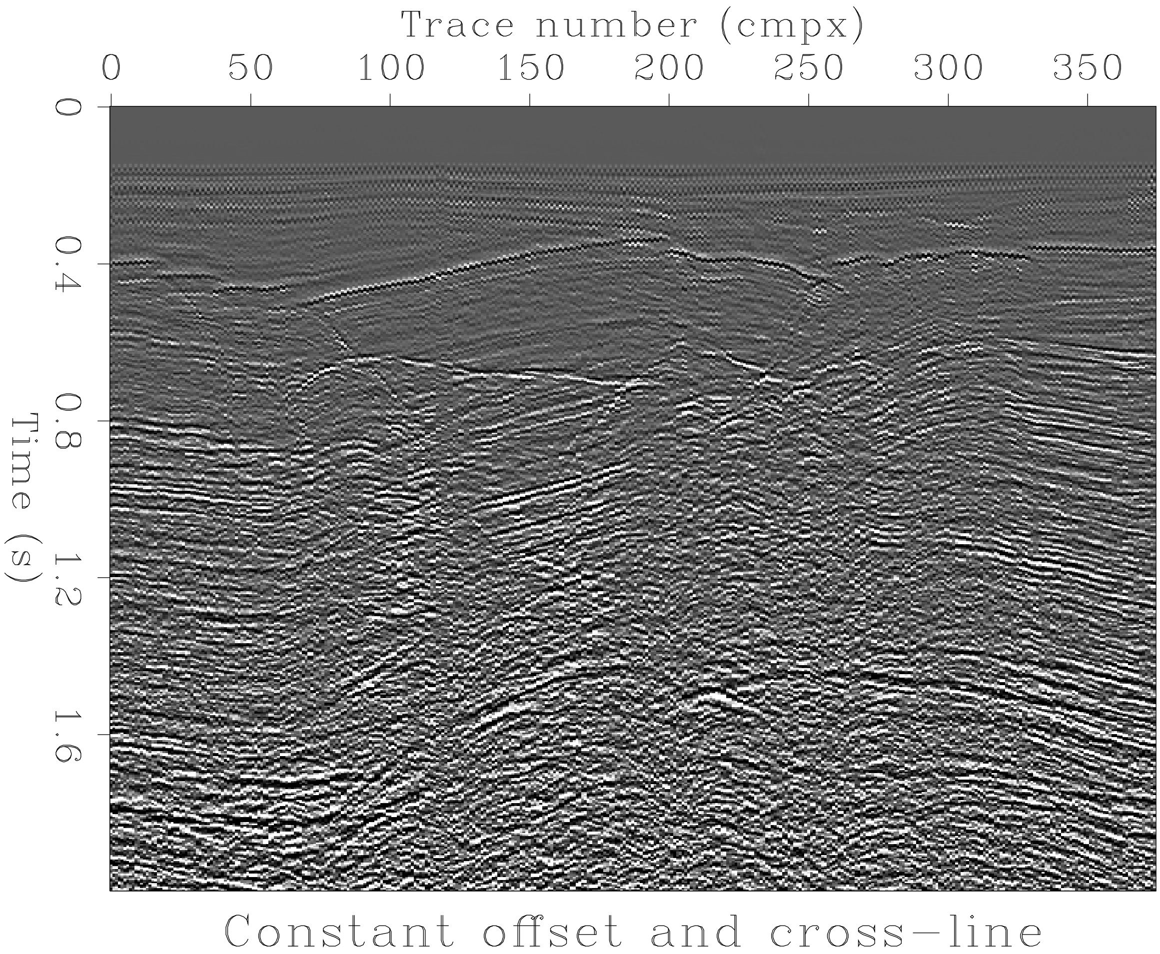

Figure 7: Constant offset and cross-line midpoint section biondo2-Coff [CR]

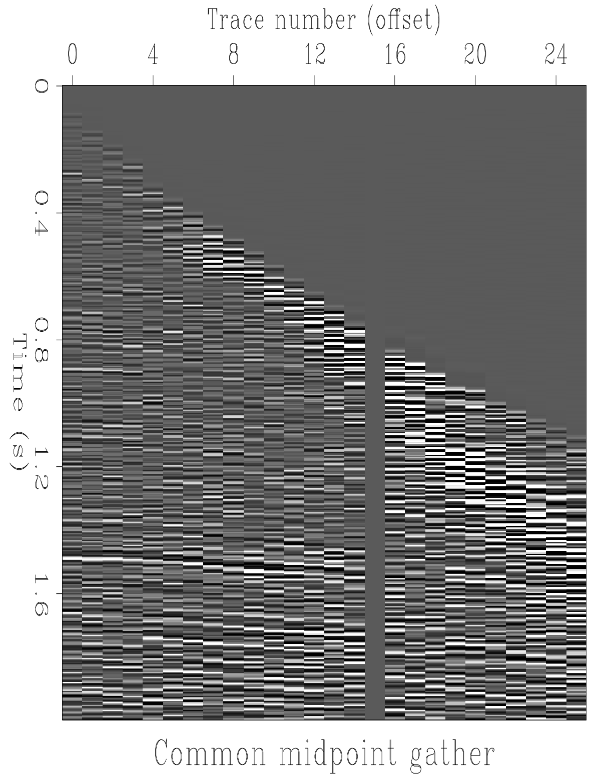

Figure 8: CDP gather biondo2-Cdp [CR]

```
integer num_grid_axes,tot_grid_cells,n_in_bin
integer n_grid(num_grid_axes)

integer f(num_grid_axes),j(num_grid_axes) ! notice automatic allocation
integer grid_val(n_in_bin,tot_grid_cells) ! notice automatic allocation
real fold(tot_grid_cells) ! notice automatic allocation

f=0 ! notice array syntax for initilization
j=1

! get the pointers to header record number for the whole grid
if (sep_get_grid_window('in',num_grid_axes,n_grid,n_grid,f,j,grid_val) /= 0) &
        call seperr('error in sep_get_grid_window')

where (grid_val > 0)
   grid_val =1 !There is a trace if the pointer is non-negative
elsewhere
   grid_val =0 !There is NO trace if the pointer is negative
end where

fold=float(sum(grid_val,dim=1)) !notice use of reduction intrinsic SUM

if(srite('out',fold,tot_grid_cells*4) /= tot_grid_cells*4) &
         call seperr('error in srite')

return
end
```

This subroutine is the main subroutine of a SEPlib90 application program (`Fold90`). Figure 9 shows the fold values for cross-line of the data set computed using `Fold90`.

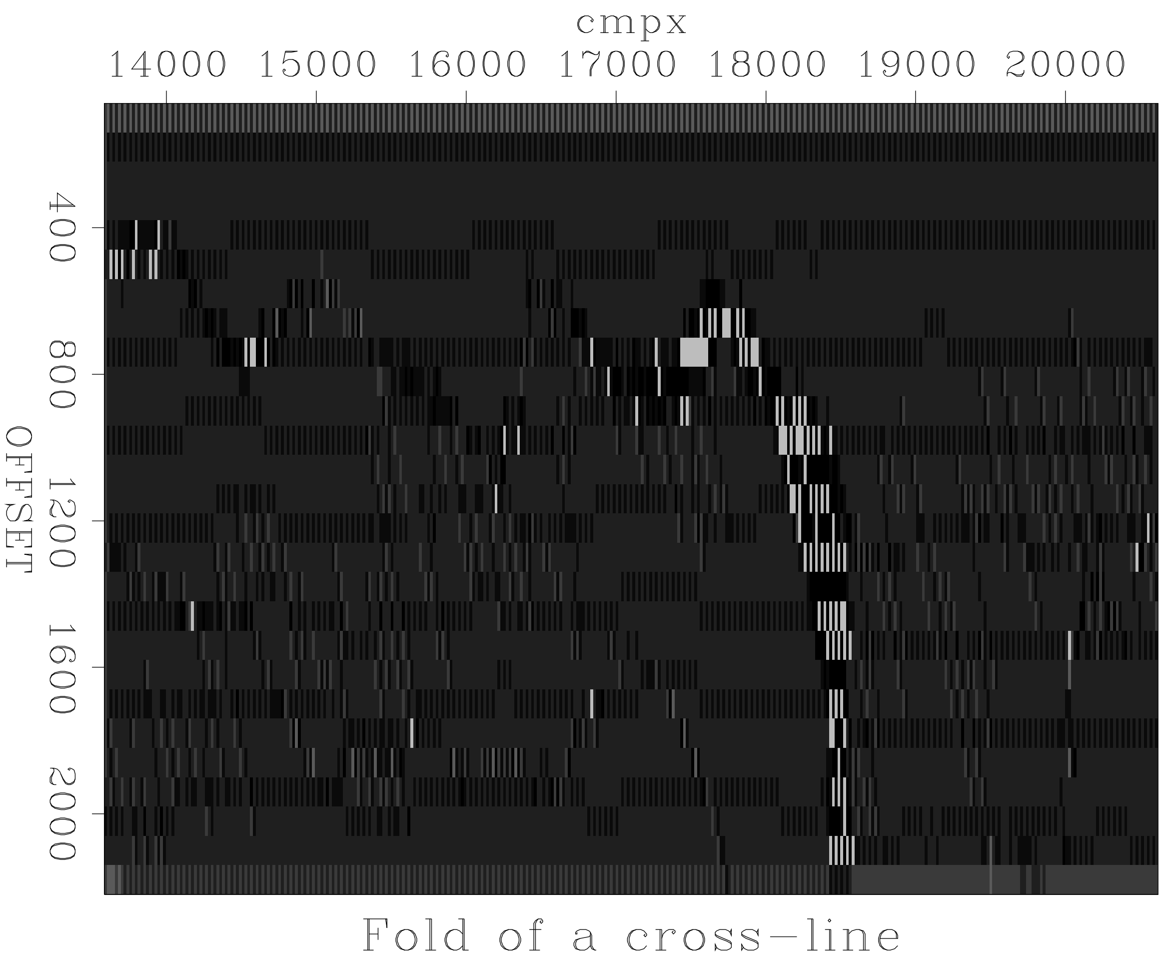

Figure 9: Fold values for constant cross-line midpoint biondo2-Fold [CR]

SUMMARY

SEPlib90 is a new powerful tool in the hands of SEP researchers to develop and test methods for processing irregularly sampled data. It overcomes the limitations of SEPlib77 by including the trace headers information in the data format and enabling easy and efficient access to such information. Further, it exploits existing regularity in data geometry by associating with the data traces a uniformly sampled multi-dimensional grid and by providing tools to efficiently access subsets of the data set based on this gridding information.

ACKNOWLEDGMENTS

We would like to thank Conoco and its partners BP and Mobil, for agreeing to release the real data that we showed in this paper.

REFERENCES

Barry, K. M., Cavers, D. A., and Kneale, C. W., 1975, Report on recommended standards for digital tape formats: Geophysics, **40**, no. 2, 344–352.

Claerbout, J. F., 1991, SEPlib and SEP software-computer: SEP–**71**, 283–286.

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

Dellinger, J., and Tálas, S., 1992, A tour of SEPlib for new users: SEP–**73**, 461–502.

Nichols, D., Karrenbach, M., and Urdaneta, H., 1994, What's new in SEPLIB?: SEP–**82**, 257–264.