*Stanford Exploration Project, Report 82, May 11, 2001, pages 1–207*

# What's new in SEPLIB?

*Dave Nichols, Martin Karrenbach, and Hector Urdaneta*[1]

**ABSTRACT**

The CD-ROM version of this report is distributed with a new version of the SEP software "SEPLIB" on it. The new version has a more robust Input/Ouput (I/O) system that is extendible to support new types of I/O. It also unifies the behavior of all input and output files. The distribution contains experimental support for a geometry database to permit the use of non-uniformly sampled data.

## INTRODUCTION

The SEPLIB software package has been developed over the last fifteen years at SEP (**?**) It is used for almost all the research at SEP and it is supported on all the machines that we own (and those we have owned in the past). It is designed using the Unix philosophy of providing many small programs that each do a simple job. The individual programs can be combined by using the facilities of Unix to pipe the commands together or by writing shell scripts that perform sequences of commands (**?**). Over the last few years we have made some changes in SEPLIB to support new types of I/O (e.g. fast parallel reads and writes on the Thinking Machines CM-5) and to make the software more flexible. We have also added support for more variants of the Unix operating system. The following machines and operating systems (OS) are actively supported:

- Sun, SunOS 4.x
- HP-700, HP-UX 9.x
- IBM RS6000, AIX 4.x
- CM-5, CMOST
- Cray YMP, C-90, Unicos.

In the past SEPLIB was used on the following machines and the distribution still contains the configuration files.

- DEC mips, Ultrix 4.x

[1]**email:** dave@sep.stanford.edu,martin@sep.stanford.edu,hector@sep.stanford.edu

- DEC-alpha, OSF-1
- Convex C series, Convex-OS
- SGI, IRIX

## CHANGES TO SEPLIB I/O

The SEPLIB package was originally created to read header files from standard input (stdin) and write header files to standard output (stdout). The input and output can be connected to regular files or to pipes. If the header is written to a regular file then the large data file is written in a different directory and its location is noted in the header file. If the header is written to another process down a pipe, then the data file is written down the same pipe. Subsequently the ability to write to other output files,named “auxiliary outputs”, was added. However, this was implemented in completely separate code. The auxiliary files could only be regular files. In the new version of SEPLIB the I/O routines have been unified. The same code is used for all input and output files. This allows all the files to read and write from one of three entities:

1. a file.
2. a pipe to another process.
3. a network socket connection.

In addition, the option has been added to allow the data to be written to the same file as the header data, though the file now begins as ASCII and ends as binary. This permits the creation of location independent files that can be easily copied as a single unit or written to tape as a single file.

### Examples of new SEPLIB I/O

Assume that we have written a program called Myprog that contains the following lines:

```
nread = sreed('in',data,nbytes)
nread = sreed('vfile',velocity,nbytes)
............
nwrite = srite('out',data,nbytes)
nwrite = srite('snaps',snapshots,nbytes)
```

This program reads from stdin `(tag="in")` and an auxiliary input `(tag="vfile")`. It writes to stdout `(tag="out")` and to an auxiliary output `(tag="snaps")`. These tag names can be used on the command name to change the nature of the input and output. Note that:

1. The file specified on the command line specifies the header file name. The data is actually read from the data file which is defined in the header file.
2. Tags `"in"` and `"out"` have special command line name `"stdin"` and `"stdout"` because `"in"` and `"out"` have other meanings for historical reasons.

Following are some examples of commands using this program: `Myprog <data.H >image.H`

- Read tag "in" from stdin, connected to header file "data.H".
- Write tag "out" to stdout, connected to header file "image.H".

```
Myprog <data.H vfile=vels.H snaps=tmp.H >image.H
```

- Read tag "in" from stdin, connected to header file "data.H".
- Write tag "out" to stdout, connected to header file "image.H".
- Read tag "vfile" from header file "vels.H" .
- Write tag "snaps" to header file "tmp.H" .

```
Myprog stdin=data.H stdout=image.H
```

- Read tag "in" from header file "data.H".
- Write tag "out" to header file "image.H".

```
Myprog <data.H | Wiggle | Tube
```

- Write tag "out" to stdout, a pipe to "Wiggle and Tube".

```
Myprog <data.H snaps="|Wiggle|Tube" >image.H
```

- Write tag "snaps" to a pipe to "Wiggle and Tube".

```
Spike n1=10 k1=1 | Myprog >image.H
```

- Read tag "in" from stdin, a pipe from "Spike".

```
Myprog <data.H vfile="Spike n1=10 | Add add=10.  |" >image.H
```

- Read tag "vfile" from a pipe from "Spike and Add".

```
Myprog <data.H >image.H out=stdout
```

- Write tag "out" to file "image.H", data follows header to the same file.

```
Myprog <data.H isoutstream=y | cat >image.H
```

- Same as above "isoutstream=y" makes it write the output in the portable "stream" format, even though it is writing to a pipe.

```
Myprog <data.H isoutstream=y | dd bs=1024 of=/dev/st0
```

- Write tag "out" to tape in stream format.

```
dd if=/dev/st0 bs=1024 | Myprog >image.H
```

- Read tag "in" from tape in stream format.

```
Myprog <data.H stdout=":3001"
```

- More esoteric. Tag "out" is connected to a socket on this machine.
  It opens port 3001 and waits for a connection.

```
Myprog stdin="spur:3001" >image.H
```

- More esoterica. Tag "in" is connected to a socket.
  It attempts to connect to port 3001 on machine "spur"

### Modular data I/O

The internals of SEPLIB have been modified to provide a more expandable mechanism for adding new types of I/O for data files. Rather than calling I/O routines directly, the routines `sreed(),srite(),sseek()`, and `ssize()` call routines specified by function pointers. This allows the programmer to change the mechanism for the physical I/O by simply writing a new set of routines and changing the function pointers. The current library supports four types of I/O.

1. Buffered I/O using the Unix routines fread, fwrite, fseek, etc. This is used when the header and data are written to the same file or if the data follows the header down a pipe or socket.
2. Parallel I/O to the connection machine disk array from a CM-Fortran parallel array. This is used when the user specifies parallel I/O by calling the function `set_hpf_io( tagname ) .`

3. Multiple file I/O. This is an experimental extension that allows the data to be spread over many physical files but still presenting it to the user as a single logical file. It is used when the output file name or datapath are semi-colon separated lists of filenames or directories.

4. Unbuffered I/O using the unix routines read, write, lseek, etc. The default mechanism for writing to a regular file.

## GEOMETRY HANDLING

The notion of a SEPLIB data cube as a regularly gridded cube has been extended to allow for a dataset with irregular spacing. This functionality is contained in a small subroutine library `libsepattr.a`. Linking with this library ( -lsepattr in the link command) allows us to retrieve "header" values for a given trace. When we talk about SEPLIB cubes in this paragraph, we actually mean SEPLIB hypercubes of arbitrary dimensions. When designing this extension to the normal SEPLIB dataset usage, we had two issues in mind. First, to keep all information pertaining to a specific trace separate from the trace itself. That would allow us to operate on the attributes without having to move the data around at the same time and would also allow for flexible implementation of that trace attribute information in the form of databases or flat files. The user then would communicate his action through a common interface without having to know the physical representation of the attributes. Second, we wanted to be able to operate with normal SEPLIB programs on that attribute information. The most common task would be to check consistency of values or their spatial distribution, such as stacking-charts. Currently, each trace has a unique trace identifier, which is its real coordinates in the SEPLIB hypercube. The "header (attribute) database" has logically the same dimensionality as the original SEPLIB data cube. The physical dimensionality need not be the same. This can be seen, for example, when an original SEPLIB dataset is reduced by a windowing process, but the "header data base" is not. The trace in the windowed SEPLIB data cube still has the same coordinates pointing to the correct information in the "header data base" file. This allows us to keep just one copy of the header information through a sequence of different processing steps without duplicating information unnecessarily. Only when sorted is it necessary to create a new header database. Header values are stored in the machine-independent representation "xdr-format", and a header can take any "xdr" format. The link of a SEPLIB dataset with its associated header data base (file) is determined by SEPLIB parameter argument `attrfile="any-filename"` which can occur on the command line or in the history file of the SEPLIB data set. When using the `libsepattr` utilities, this link is activated and all necessary information is retrieved. The header database file "any-filename" is currently another SEPLIB file that contains the additional information:

```
hdrkey1="any-key-name"   hdrfmt1="any-xdr-format"
hdrkey2="any-key-name"   hdrfmt2="any-xdr-format"
hdrkey3="any-key-name"   hdrfmt3="any-xdr-format"
      .                              .
      .                              .
```

. .

The consecutively numbered header keys are counted automatically and the number of keys is not limited. The current formats are `xdr_float`, `xdr_int`, `xdr_short`, `xdr_byte`, `xdr_opaque`, `xdr_native`. If a user wants to utilize the header information in his program, he has to call `attrin("tagname")` to open the input database file. The key values are retrieved by either specifying a keyname or its location (index) in the database. To get a value by name `attrget(keyname,identifier,value)`, where `keyname` is a string containing the keyname and `identifier` is an array of coordinates of that trace; the variable `value` contains the key's value on return. The index of a key can be obtained by the function `idx = attrindex (keyname)`. Once you know the index of a key you can get the `keyname` using `attrkeyname(index,keyname)` and its type by `attrkeytype(index,keytype)`, where `keytype` is a string. Another way to retrieve the header value is to look up by index `attrgetbyindex(keyindex, identifier, value)`. The user can modify and write out a new header database by calling `attrout("tag")` and put values by `attrput(keyname,identifier,value)`. All of the subroutines will return `0` on successful completion. More details can be found in the manual pages describing each subroutine. For convenience, some utility programs are provided and can be found in the subdirectory `cube/attrutil`. The most commonly used are `Segy2` which is used to read in a SEGY dataset and to convert it to this extended SEPLIB dataset. `Cubify` sort according to header names and regularize (bin) the datset to be a padded hypercube. `Sort` sorts such a dataset (either header database only or dataset only or both) according to key values. Diagnostic programs that can also be used as example programs are `Dumpkeys, Getdblist, Getdbindex, Getdbval` and `Putdbval`. The above described method of dealing with trace attribute values has been used extensively by a few people within SEP when dealing with real datasets and is still under development and improvement.

## OTHER CHANGES TO SEPLIB

- All documentation is now in the source files. A script to automatically generate manual pages is supplied.
- A new routine to cache a piped file is provided, make_unpipe(). The often-used program `Byte` uses this routine, so you can now conveniently pipe to `| Byte`.
- New routines `sreed_raw(), srite_raw(), sseek(), ssize()` are added. All seplib programs should use `sreed, srite, srite_raw, sreed_raw, sseek, ssize`. This is a side effect of the more modular I/O. The return values from auxin/auxout should only be checked to discover any errors and should not be used directly.
- Old programs that use `reed, rite, lseek, fread` etc. should be changed.
- All distributed programs have been modified to obey these new rules.
- Shared libraries are supported on SUN-OS. This required the splitting of SEPLIB libraries into a "C" library, libsep.a, and a "Fortran" library, libsepf.a. Fortran programs should link with "-lsepf -lsep". C programs only need "-lsep".

- New routine `auxinout()` opens a tag for input and output. `auxin()` now opens for input only and `auxout()` for output only. It must be the first function or subroutine called with the tag.

## SUMMARY

The SEPLIB software package, developed over the last fifteen years, is used for almost all the research at SEP. The basic data structure is a gridded hypercube and has now been extended to allow for irregular traces which are still arranged as a hypercube. SEPLIB has not only had a face-lift, but its internals have also been redesigned to provide more flexibility and efficiency on today's high performance vector and parallel unix computers, as well as workstations.

## REFERENCES