Stanford Exploration Project, Report SERGEY, May 1, 2000, pages 619–626

"SEP" module: A Fortran-90 interface to SEPlib

Sergey Fomel[1]

ABSTRACT

A simplified interface to SEPlib is implemented with a Fortran-90 module.

INTRODUCTION

SEPlib is a software package, developed at Stanford Exploration Project for all-purpose data processing (Claerbout, 1991a,b; Dellinger and Tálas, 1992; Nichols et al., 1994). SEPlib is based on a special data format. The data is stored in a architecture-independent binary format and can be retrieved in a multi-dimensional array (cube). The description of the data is provided in a separate file, which is called the "history" file. The history file is a readable plain text (ASCII) file, which contains a pointer to the binary data (`in=` tag) and information about the cube dimensions (`n1=`, `n2=`, etc), the data type (`esize=`), the coordinate axis (`o1=`, `d1=`, `label1=`, etc), and the processing history. A C library (callable from Fortran) allows one to write and read data arrays in SEPlib files as well as read parameters from the command line and from history files.

The SEPlib library is a powerful and portable software tool, which has been in a heavy use at SEP in the past fifteen years. However, the limitations of the C programming language make the SEPlib interface more complicated than an ideal data and parameter interface should be. As a result, the main programs, operating on SEPlib files are often cluttered with data-access function calls, prone to typing errors. Recent progress in computer languages, in particular the development of object-oriented programming, allows us to build a simpler and more convenient interface on top of the existing software.

In this paper, I describe a Fortran-90 module, which implements this idea. Although Fortran-90 is not regarded as a "full-blooded" object-oriented language , such as C++ or Java, it has a number of *object-based* features (Gray and Roberts, 1997), such as operator overloading, optional arguments, private variables, etc., which facilitate an optimal interface design. In our experience, the Fortran-90 interface, implemented in the `sep` module makes writing SEPlib programs faster and the resulting code shorter and more readable. Additionally, since the implementation details are hidden from the user, they can be improved or changed in the future without affecting the interface itself.

A recent extension to SEPlib is SEPlib90 (also known as SEPlib3D), which provides an additional support for irregular data geometries, typical in 3-D seismic data acquisition (Biondi

[1]**email:** sergey@sep.stanford.edu

et al., 1996). The `sep` module currently does not support SEPlib90 functions, although it is possible to add this support in the future. An object-based interface to SEPlib90, called `sepf90` was created by Clapp and Crawley (1996b). The `sepf90` interface maps a SEPlib90 data file to a specially defined data type. I avoided the natural temptation to use a similar approach in the `sep` module, because it would take some flexibility out of the original SEPlib interface.

USING THE MODULE

To use the `sep` module, simply put

```
use sep
```

in your program. All operations with parameter and header files should be enclosed in two subroutine calls:

```
call sep_init ()
```

and

```
call sep_close ()
```

Essentially, `sep_init` just calls `initpar`, and `sep_close` calls `hclose`, but the new names make them look more like a constructor-destructor pair, typical for the object-based style of programming. The calls to data access functions (`sep_read` and `sep_write`), described below, can be placed anywhere in the program. My typical main-program style is

1. get parameters
2. allocate storage
3. read the input
4. do processing (sometimes by using other modules)
5. write the output
6. free storage

Sometimes it is more convenient to do it in a different order (for example, when some of your parameters are defined in the processing step). An example main program is `Clip` on the current page. Next sections explain the function interface used in this example.

```
program Clip
  use sep

  implicit none
  integer                          :: n1, n2, i2
  real                             :: c
  real, dimension (:), allocatable :: data

  call sep_init()
  call from_history (n1,n2,compress=.true.)
  call from_par ("clip", c)
  call sep_close ()

  allocate (data (n1))
  do i2 = 1, n2
     call sep_read (data)
     where (data > c) data = c
     call sep_write (data)
  end do
  deallocate (data)

  call exit (0)
end program Clip
```

PARAMETER HANDLING

Parameters can be read from the command line or a parameter file, specified with `par=parfile` on the command line. To read a parameter, use

```
call from_par ("clip",c)
```

where `clip` is the parameter name, and `c` is a variable that gets the parameter value. For example, if the command line (or a parameter file) has `clip=0.75`, the variable `c` gets assigned the value of `0.75`. By making use of function overloading, we can avoid specifying an additional parameter for the data type. Which format to use for reading the parameter value is determined directly from the type of variable `c`. For example, if `c` is a character string, it will be assigned the value of `"0.75"` (a string of four characters). A call to `from_par` shown above will result in an error message if the parameter was not specified by the user. You can avoid this situation by specifying a default value for `c`, as follows:

```
call from_par ("clip",c,0.8)
```

Now if `clip=` is not present in the parameter list, `c` will be assigned the value of `0.8`. The types of the second and third parameter in `from_par` should match (if they don't, you will get a compile-time error). All parameter values, read by `from_par`, are also written to the output history file. Another common source of parameters is the input history file. To read a parameter from the history file, use

```
call from_history ("clip",c)
```

Again, the reading format is defined by the type of `c` (real, integer, or character), and you can specify an optional value with something like

```
call from_history ("clip",c,0.8)
```

The most important information that we normally read from a history file is the data size. We can read the size parameters in the normal manner

```
call from_history ("n1",nt)
call from_history ("n2",nx)
call from_history ("n3",ny)
```

but the `sep` module provides also a handy shortcut:

```
call from_history (nt,nx,ny)
```

is equivalent to the three calls above. In this form of `from_history`, only the first parameter is mandatory. Thus, if you use

```
call from_history (nt,nx)
```

it will read only the first two dimensions from the history file. This form of `from_history` also reads the `esize` parameter and checks if it has the right value. By default, it assumes `esize=4`, which corresponds to real-valued data. If you work with a different type of data, just add the `esize` parameter to the call. For example,

```
call from_history (nt,nx,esize=8)
```

will work with complex data. Another useful parameter is `compress`. Sometimes it doesn't matter for your processing what the cube dimensions are. An example is program `Clip` on page 620, where it is easier to treat the data as a one or two-dimensional array. If you use

```
call from_history (n1,n2,compress=.true.)
```

or just

```
call from_history (n1,compress=.true.)
```

it will read all[2] dimensions, specified in the history file and "compress" them to one or two for easier processing. Sometimes, you may need more than one input to your program. A typical example is a migration program, which reads the data cube from one file, and the velocity cube from another file. To read parameters from another history file, use

[2] up to 99

```
call from_aux ("velocity","d1",dz)
```

The syntax of `from_aux` is exactly the same as that of `from_history` except for the additional first parameter, which specifies the label of the auxiliary file. In the call above, it is assumed that the command line (or parameter file) has something like `velocity=vel.H`. How can you test if an auxiliary file was actually specified? Use

```
if (exist_file ("velocity")) then
```

The function `exist_file` accepts one argument and returns a logical value. It can be used equally well for testing parameters (other than file names), although the latter is more commonly done with default arguments. Another parameter-access subroutine is `from_old` . This subroutine works exactly like `from_history` except that it reads not the latest value of the parameter, but the one before it. This has to do with the old-fashioned way of coding adjoint operators, which is hardly ever used in our current practice. A more useful variation is `from_either` . This one allows you to search for the parameter either in the history file or on the command line (with latter having a priority). The interface syntax is similar to that of `from_history`. Writing a parameter to the output history file is done with

```
call to_history ("n1",nt)
```

How do you write to another history file? Just add an optional parameter with the file tag[3]

```
call to_history ("n1",nt,"velocity")
```

SUPPORT FOR N-DIMENSIONAL PROCESSING

Helical filtering (Claerbout, 1997a) opens new possibilities for multidimensional filtering applications. In particular, it becomes possible to write completely dimension-independent programs for deconvolution, missing data interpolation, least-square signal and noise separation, etc. (Claerbout, 1997b). Of course, going beyond the traditional one, two, and three dimensions has not only the theoretical value. Prestack seismic data are naturally five-dimensional (with a "3-D" acquisition), and additonal dimensions (depth, velocity, etc.) are often added by processing. Besides, dimension-independent programming avoids a large amount of an error-prone code duplication, required for going from one to two to three dimensions. The `sep` module provides some additional functionality to support N-dimensional programming. The function `sep_dimension` takes the file tag (defaulted to `"in"` - the standard input) and returns the effective number of dimensions in the SEPlib file. Additionaly, `from_par`, `from_history`, `to_history`, and other parameter handling routines accept not only scalar arguments, but also arrays of values. If `iarr` is an array of integer values, then

[3]Note that file name is the first parameter in `from_aux` but the last parameter in `to_history`. This avoids the necessity of writing a `to_aux` subroutine.

```
call from_par ("arr",iarr)
```

whill fill `iarr` with parameter values `1, 5, 3`, when the command line (or a parameter file) has either a comma-separated list `arr=1,5,3` or distinct values `arr1=1`, `arr2=5`, `arr3=3`. This convention can be handy for reading all the dimensions from a file:

```
allocate (n (sep_dimension ()))
call from_history ("n",n)
```

It also works with `to_history`: short arrays are written to the history file as comma-separated lists, while long arrays are expanded as distinct parameters. A suite of N-dimensional programs (using the `sep` module) accompany the current revision of Claerbout's *Geophysical Estimation Mapping*.

DATA ACCESS

In the `sep` module, all data I/O is concentrated in two subroutines: `sep_read` and `sep_write` . In the simplest case, you just need to allocate an array of proper type, shape, and dimensions, and then use

```
call sep_read (array)
```

or

```
call sep_write (array)
```

Here `array` can be one-, two- or three-dimensional, and belong to real or complex type. Internally, the proper number of binary bites is calculated with the Fortran-90 `size` function. By default, `sep_read` reads from SEPlib's standard input, and `sep_write` writes to SEPlib's standard output. To use a different file, just add an optional argument:

```
call sep_read (array,"velocity")
```

or

```
call sep_write (array,"velocity")
```

If you return now to program `Clip` on page 620, you should be able to understand all parts of it. A brief index of functions and subroutines is enclosed at the end of the paper.

CONCLUSIONS

Fortran-90 allows us to design a simple functional interface for SEPlib. The interface has been implemented in the `sep` module. Depending on the compiler, the Fortran-90 interface may slow down the performance of I/O operations a little bit because of the function-call overhead. This slowdown is insignificant in comparison with the development time the new interface saves us. Moreover, separating the interface and the implementation allows us to work on improving both these parts independently.

ACKNOWLEDGMENTS

I acknowledge Bob Clapp and Sean Crawley for their work in introducing Fortran-90 to SEP (Clapp and Crawley, 1996a).

REFERENCES

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

Claerbout, J. F., 1991a, Introduction to Seplib and SEP utility software: SEP–**70**, 413–436.

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

Claerbout, J., 1997a, Multidimensional recursive filters via a helix: SEP–**95**, 1–13.

Claerbout, J. F., 1997b, Geophysical exploration mapping: Environmental soundings image enhancement: Stanford Exploration Project.

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: SEP–**93**, 293–304.

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

Gray, M. G., and Roberts, R. M., 1997, Object-based programming in Fortran-90: Computers in Physics, **11**, no. 4, 355–361.

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

INDEX