Guide to RSF API

Sergey Fomel[1]

ABSTRACT

This guide explains the RSF programming interface.

INTRODUCTION

To work with RSF files in your own programs, you may need to use an appropriate programming interface. We will demonstrate the interface in different languages using a simple example. The example is a clipping program. It reads and writes RSF files and accesses parameters both from the input file and the command line. The input is processed trace by trace. This is not necessarily the most efficient approach[2] but it suffices for a simple demonstration.

C INTERFACE

The C clip function is listed below.

```
1  /* Clip the data. */
2
3  #include <rsf.h>
4
5  int main(int argc, char* argv[])
6  {
7      int n1, n2, i1, i2;
8      float clip, *trace;
9      sf_file in, out; /* Input and output files */
10
11     /* Initialize RSF */
12     sf_init(argc,argv);
13     /* standard input */
14     in  = sf_input("in");
15     /* standard output */
16     out = sf_output("out");
17
```

[1]**e-mail:** sergey.fomel@beg.utexas.edu

[2]Compare with the library clip program.

```
18      /* check that the input is float */
19      if (SF_FLOAT != sf_gettype(in))
20          sf_error("Need float input");
21
22      /* n1 is the fastest dimension (trace length) */
23      if (!sf_histint(in,"n1",&n1))
24          sf_error("No n1= in input");
25      /* leftsize gets n2*n3*n4*... (the number of traces) */
26      n2 = sf_leftsize(in,1);
27
28      /* parameter from the command line (i.e. clip=1.5 ) */
29      if (!sf_getfloat("clip",&clip)) sf_error("Need clip=");
30
31      /* allocate floating point array */
32      trace = sf_floatalloc (n1);
33
34      /* loop over traces */
35      for (i2=0; i2 < n2; i2++) {
36
37          /* read a trace */
38          sf_floatread(trace,n1,in);
39
40          /* loop over samples */
41          for (i1=0; i1 < n1; i1++) {
42              if      (trace[i1] >  clip) trace[i1]= clip;
43              else if (trace[i1] < -clip) trace[i1]=-clip;
44          }
45
46          /* write a trace */
47          sf_floatwrite(trace,n1,out);
48      }
49
50
51      exit(0);
52  }
```

Let us examine it in detail.

```
3   #include <rsf.h>
```

The include preprocessing directive is required to access the RSF interface.

```
9       sf_file in, out; /* Input and output files */
```

RSF data files are defined with an abstract `sf_file` data type. An abstract data type means that the contents of it are not publicly declared, and all operations on

`sf_file` objects should be performed with library functions. This is analogous to `FILE *` data type used in `stdio.h` and as close as C gets to an object-oriented style of programming (Roberts, 1998).

```
11      /* Initialize RSF */
12      sf_init(argc,argv);
```

Before using any of the other functions, you must call `sf_init`. This function parses the command line and initializes an internally stored table of command-line parameters.

```
13      /* standard input */
14      in  = sf_input("in");
15      /* standard output */
16      out = sf_output("out");
```

The input and output RSF file objects are created with `sf_input` and `sf_output` constructor functions. Both these functions take a string argument. The string may refer to a file name or a file tag. For example, if the command line contains `vel=velocity.rsf`, then both `sf_input("velocity.rsf")` and `sf_input("vel")` are acceptable. Two tags are special: `"in"` refers to the file in the standard input and `"out"` refers to the file in the standard output.

```
18      /* check that the input is float */
19      if (SF_FLOAT != sf_gettype(in))
20          sf_error("Need float input");
```

RSF files can store data of different types (character, integer, floating point, complex). We extract the data type of the input file with the library `sf_gettype` function and check if it represents floating point numbers. If not, the program is aborted with an error message, using the `sf_error` function. It is generally a good idea to check the input for user errors and, if they cannot be corrected, to take a safe exit.

```
22      /* n1 is the fastest dimension (trace length) */
23      if (!sf_histint(in,"n1",&n1))
24          sf_error("No n1= in input");
25      /* leftsize gets n2*n3*n4*... (the number of traces) */
26      n2 = sf_leftsize(in,1);
```

Conceptually, the RSF data model is a multidimensional hypercube. By convention, the dimensions of the cube are stored in `n1=`, `n2=`, etc. parameters. The `n1` parameter refers to the fastest axis. If the input dataset is a collection of traces, `n1` refers to the trace length. We extract it using the `sf_histint` function (integer parameter from history) and abort if no value for `n1` is found. We could proceed in a similar fashion, extracting `n2`, `n3`, etc. If we are interested in the total number of traces, like in the clip example, a shortcut is to use the `sf_leftsize` function. Calling `sf_leftsize(in,0)`

returns the total number of elements in the hypercube (the product of `n1`, `n2`, etc.), calling `sf_leftsize(in,1)` returns the number of traces (the product of `n2`, `n3`, etc.), calling `sf_leftsize(in,2)` returns the product of `n3`, `n4`, etc. By calling `sf_leftsize`, we avoid the need to extract additional parameters for the hypercube dimensions that we are not interested in.

```
28      /* parameter from the command line (i.e. clip=1.5 ) */
29      if (!sf_getfloat("clip",&clip)) sf_error("Need clip=");
```

The clip parameter is read from the command line, where it can be specified, for example, as `clip=10`. The parameter has the `float` type, therefore we read it with the `sf_getfloat` function. If no `clip=` parameter is found among the command line arguments, the program is aborted with an error message using the `sf_error` function.

```
31      /* allocate floating point array */
32      trace = sf_floatalloc (n1);
```

Next, we allocate an array of floating-point numbers to store a trace with the library `sf_floatalloc` function. Unlike the standard `malloc` the RSF allocation function checks for errors and either terminates the program or returns a valid pointer.

```
34      /* loop over traces */
35      for (i2=0; i2 < n2; i2++) {
36
37          /* read a trace */
38          sf_floatread(trace,n1,in);
39
40          /* loop over samples */
41          for (i1=0; i1 < n1; i1++) {
42              if      (trace[i1] >  clip) trace[i1]= clip;
43              else if (trace[i1] < -clip) trace[i1]=-clip;
44          }
45
46          /* write a trace */
47          sf_floatwrite(trace,n1,out);
48      }
```

The rest of the program is straightforward. We loop over all available traces, read each trace, clip it and right the output out. The syntax of `sf_floatread` and `sf_floatwrite` functions is similar to the syntax of the C standard `fread` and `fwrite` function except that the type of the element is specified explicitly in the function name and that the input and output files have the RSF type `sf_file`.

Compiling

To compile the `clip` program, run

```
cc clip.c -I$RSFROOT/include -L$RSFROOT/lib -lrsf -lm
```

Change `cc` to the C compiler appropriate for your system and include additional compiler flags if necessary. The flags that RSF typically uses are in `$RSFROOT/share/madagascar/etc/config.py`.

C++ INTERFACE

The C++ clip function is listed below.

```
/* Clip the data. */

#include <valarray>
#include <rsf.hh>

int main(int argc, char* argv[])
{
    sf_init(argc,argv); // Initialize RSF

    iRSF par(0), in; // input parameter, file
    oRSF out;        // output file

    int n1, n2;      // trace length, number of traces
    float clip;

    in.get("n1",n1);
    n2=in.size(1);

    par.get("clip",clip); // parameter from the command line

    std::valarray<float> trace(n1);

    for (int i2=0; i2 < n2; i2++) { // loop over traces
        in >> trace; // read a trace

        for (int i1=0; i1 < n1; i1++) { // loop over samples
            if      (trace[i1] >  clip) trace[i1]=clip;
            else if (trace[i1] < -clip) trace[i1]=-clip;
        }

```

```
31          out << trace; // write a trace
32      }
33
34      exit(0);
35  }
```

Let us examine it line by line.

```
4   #include <rsf.hh>
```

Including "`rsf.hh`" is required for accessing the RSF C++ interface.

```
8       sf_init(argc,argv); // Initialize RSF
```

A call to `sf_init` is required to initialize the internally stored table of command-line arguments.

```
10      iRSF par(0), in; // input parameter, file
11      oRSF out;        // output file
```

Two classes: `iRSF` and `oRSF` are used to define input and output files. For simplicity, the command-line parameters are also handled as an `iRSF` object, initialized with zero.

```
16      in.get("n1",n1);
17      n2=in.size(1);
```

Next, we read the data dimensions from the input RSF file object called `in`: the trace length is a parameter called "`n1`" and the number of traces is the size of `in` remaining after excluding the first dimension. It is extracted with the `size` method.

```
19      par.get("clip",clip); // parameter from the command line
```

The clip parameter should be specified on the command line, for example, as `clip=10`. It is extracted with the `get` method of `iRSF` class from the `par` object.

```
21      std::valarray<float> trace(n1);
```

The trace object has the single-precision floating-point type and is a 1-D array of length `n1`. It is declared and allocated using the `valarray` template class from the standard C++ library.

```
23      for (int i2=0; i2 < n2; i2++) { // loop over traces
24          in >> trace; // read a trace
25
26          for (int i1=0; i1 < n1; i1++) { // loop over samples
27              if      (trace[i1] >  clip) trace[i1]=clip;
```

```
28              else if (trace[i1] < -clip) trace[i1]=-clip;
29          }
30
31          out << trace; // write a trace
32      }
```

Next, we loop through the traces, read each trace from `in`, clip it and write the output to `out`.

Compiling

To compile the C++ program, run

```
c++ clip.cc -I$RSFROOT/include -L$RSFROOT/lib -lrsf++ -lrsf -lm
```

Change `c++` to the C++ compiler appropriate for your system and include additional compiler flags if necessary. The flags that RSF typically uses are in `$RSFROOT/share/madagascar/etc/config.py`.

FORTRAN-77 INTERFACE

The Fortran-77 clip function is listed below.

```
1           program Clipit
2           implicit none
3           integer n1, n2, i1, i2, in, out
4           integer sf_input, sf_output, sf_leftsize, sf_gettype
5           logical sf_getfloat, sf_histint
6           real clip, trace(1000)
7
8           call sf_init()
9           in = sf_input("in")
10          out = sf_output("out")
11
12          if (3 .ne. sf_gettype(in))
13       &  call sf_error("Need float input")
14
15          if (.not. sf_histint(in,"n1",n1)) then
16             call sf_error("No n1= in input")
17          else if (n1 > 1000) then
18             call sf_error("n1 is too long")
19          end if
20          n2 = sf_leftsize(in,1)
```

```
21
22          if (.not. sf_getfloat("clip",clip))
23         &  call sf_error("Need clip=")
24
25          do 10 i2=1, n2
26             call sf_floatread(trace,n1,in)
27
28             do 20 i1=1, n1
29                if (trace(i1) >  clip) then
30                   trace(i1)=clip
31                else if (trace(i1) < -clip) then
32                   trace(i1)=-clip
33                end if
34  20         continue
35
36             call sf_floatwrite(trace,n1,out)
37  10      continue
38
39          stop
40          end
```

Let us examine it in detail.

```
8           call sf_init()
```

The program starts with a call to `sf_init`, which initializes the command-line interface.

```
9           in = sf_input("in")
10          out = sf_output("out")
```

The input and output files are created with calls to `sf_input` and `sf_output`. Because of the absence of derived types in Fortran-77, we use simple integer pointers to represent RSF files. Both `sf_input` and `sf_output` accept a character string, which may refer to a file name or a file tag. For example, if the command line contains `vel=velocity.rsf`, then both `sf_input("velocity.rsf")` and `sf_input("vel")` are acceptable. Two tags are special: `"in"` refers to the file in the standard input and `"out"` refers to the file in the standard output.

```
12          if (3 .ne. sf_gettype(in))
13         &  call sf_error("Need float input")
```

RSF files can store data of different types (character, integer, floating point, complex). The function `sf_gettype` checks the type of data stored in the RSF file. We make sure that the type corresponds to floating-point numbers. If not, the program is aborted with an error message, using the `sf_error` function. It is generally a good

idea to check the input for user errors and, if they cannot be corrected, to take a safe exit.

```
15          if (.not. sf_histint(in,"n1",n1)) then
16             call sf_error("No n1= in input")
17          else if (n1 > 1000) then
18             call sf_error("n1 is too long")
19          end if
20          n2 = sf_leftsize(in,1)
```

Conceptually, the RSF data model is a multidimensional hypercube. By convention, the dimensions of the cube are stored in `n1=`, `n2=`, etc. parameters. The `n1` parameter refers to the fastest axis. If the input dataset is a collection of traces, `n1` refers to the trace length. We extract it using the `sf_histint` function (integer parameter from history) and abort if no value for `n1` is found. Since Fortran-77 cannot easily handle dynamic allocation, we also need to check that `n1` is not larger than the size of the statically allocated array. We could proceed in a similar fashion, extracting `n2`, `n3`, etc. If we are interested in the total number of traces, like in the clip example, a shortcut is to use the `sf_leftsize` function. Calling `sf_leftsize(in,0)` returns the total number of elements in the hypercube (the product of `n1`, `n2`, etc.), calling `sf_leftsize(in,1)` returns the number of traces (the product of `n2`, `n3`, etc.), calling `sf_leftsize(in,2)` returns the product of `n3`, `n4`, etc. By calling `sf_leftsize`, we avoid the need to extract additional parameters for the hypercube dimensions that we are not interested in.

```
22          if (.not. sf_getfloat("clip",clip))
23         &  call sf_error("Need clip=")
```

The clip parameter is read from the command line, where it can be specified, for example, as `clip=10`. The parameter has the `float` type, therefore we read it with the `sf_getfloat` function. If no `clip=` parameter is found among the command line arguments, the program is aborted with an error message using the `sf_error` function.

```
25          do 10 i2=1, n2
26             call sf_floatread(trace,n1,in)
27
28             do 20 i1=1, n1
29                if (trace(i1) >  clip) then
30                   trace(i1)=clip
31                else if (trace(i1) < -clip) then
32                   trace(i1)=-clip
33                end if
34  20         continue
35
36             call sf_floatwrite(trace,n1,out)
```

```
37  10      continue
```

Finally, we do the actual work: loop over input traces, reading, clipping, and writing out each trace.

Compiling

To compile the Fortran-77 program, run

```
f77 clip.f -L$RSFROOT/lib -lrsff -lrsf -lm
```

Change `f77` to the Fortran compiler appropriate for your system and include additional compiler flags if necessary. The flags that RSF typically uses are in `$RSFROOT/share/madagascar/etc/config.py`.

FORTRAN-90 INTERFACE

The Fortran-90 clip function is listed below.

```
1  program Clipit
2    use rsf
3
4    implicit none
5    type (file)                          :: in, out
6    integer                              :: n1, n2, i1, i2
7    real                                 :: clip
8    real, dimension (:), allocatable :: trace
9
10   call sf_init()              ! initialize RSF
11   in = rsf_input()
12   out = rsf_output()
13
14   if (sf_float /= gettype(in)) call sf_error("Need floats")
15
16   call from_par(in,"n1",n1)
17   n2 = filesize(in,1)
18
19   call from_par("clip",clip) ! command-line parameter
20
21   allocate (trace (n1))
22
23   do i2=1, n2                  ! loop over traces
24      call rsf_read(in,trace)
```

```
25
26        where (trace >  clip) trace =  clip
27        where (trace < -clip) trace = -clip
28
29        call rsf_write(out,trace)
30      end do
31    end program Clipit
```

Let us examine it in detail.

```
2     use rsf
```

The program starts with importing the `rsf` module.

```
10    call sf_init()              ! initialize RSF
```

A call to `sf_init` is needed to initialize the command-line interface.

```
11    in = rsf_input()
12    out = rsf_output()
```

The standard input and output files are initialized with `rsf_input` and `rsf_output` functions. Both functions accept optional arguments. For example, if the command line contains `vel=velocity.rsf`, then both `rsf_input("velocity.rsf")` and `rsf_input("vel")` are acceptable.

```
14    if (sf_float /= gettype(in)) call sf_error("Need floats")
```

A call to `from_par` extracts the "`n1`" parameter from the input file. Conceptually, the RSF data model is a multidimensional hypercube. The `n1` parameter refers to the fastest axis. If the input dataset is a collection of traces, `n1` corresponds to the trace length. We could proceed in a similar fashion, extracting `n2`, `n3`, etc. If we are interested in the total number of traces, like in the clip example, a shortcut is to use the `filesize` function. Calling `filesize(in)` returns the total number of elements in the hypercube (the product of `n1`, `n2`, etc.), calling `filesize(in,1)` returns the number of traces (the product of `n2`, `n3`, etc.), calling `filesize(in,2)` returns the product of `n3`, `n4`, etc. By calling `filesize`, we avoid the need to extract additional parameters for the hypercube dimensions that we are not interested in.

```
17    n2 = filesize(in,1)
```

The clip parameter is read from the command line, where it can be specified, for example, as `clip=10`. If we knew a good default value for `clip`, we could specify it with an optional argument, i.e. `call from_par("clip",clip,default)`.

```
21    allocate (trace (n1))
22
```

```
23   do i2=1, n2                    ! loop over traces
24      call rsf_read(in,trace)
25
26      where (trace >  clip) trace =  clip
27      where (trace < -clip) trace = -clip
```

Finally, we do the actual work: loop over input traces, reading, clipping, and writing out each trace.

Compiling

To compile the Fortran-90 program, run

```
f90 clip.f90 -I$RSFROOT/include -L$RSFROOT/lib -lrsff90 -lrsf -lm
```

Change `f90` to the Fortran-90 compiler appropriate for your system and include additional compiler flags if necessary. The flags that RSF typically uses are in `$RSFROOT/share/madagascar/etc/config.py`.

PYTHON INTERFACE

The Python clip script is listed below.

```
1  #!/usr/bin/env python
2
3  import numpy
4  import m8r
5
6  par = m8r.Par()
7  inp  = m8r.Input()
8  output = m8r.Output()
9  assert 'float' == inp.type
10
11 n1 = inp.int("n1")
12 n2 = inp.size(1)
13 assert n1
14
15 clip = par.float("clip")
16 assert clip
17
18 trace = numpy.zeros(n1,'f')
19
20 for i2 in xrange(n2): # loop over traces
```

```
21      inp.read(trace)
22      trace = numpy.clip(trace,-clip,clip)
23      output.write(trace)
```

Let us examine it in detail.

```
3  import numpy
4  import m8r
```

The script starts with importing the `numpy` and `rsf` modules.

```
6  par = m8r.Par()
7  inp  = m8r.Input()
8  output = m8r.Output()
9  assert 'float' == inp.type
```

Next, we initialize the command line interface and the standard input and output files. We also make sure that the input file type is floating point.

```
11 n1 = inp.int("n1")
12 n2 = inp.size(1)
13 assert n1
```

We extract the "`n1`" parameter from the input file. Conceptually, the RSF data model is a multidimensional hypercube. The `n1` parameter refers to the fastest axis. If the input dataset is a collection of traces, `n1` corresponds to the trace length. We could proceed in a similar fashion, extracting `n2`, `n3`, etc. If we are interested in the total number of traces, like in the clip example, a shortcut is to use the `size` method of the `Input` class1. Calling `size(0)` returns the total number of elements in the hypercube (the product of `n1`, `n2`, etc.), calling `size(1)` returns the number of traces (the product of `n2`, `n3`, etc.), calling `size(2)` returns the product of `n3`, `n4`, etc.

```
15 clip = par.float("clip")
16 assert clip
```

The clip parameter is read from the command line, where it can be specified, for example, as `clip=10`.

```
20 for i2 in xrange(n2): # loop over traces
21      inp.read(trace)
22      trace = numpy.clip(trace,-clip,clip)
23      output.write(trace)
```

Finally, we do the actual work: loop over input traces, reading, clipping, and writing out each trace.

Compiling

The python script does not require compilation. Simply make sure to set `PYTHONPATH` and `LD_LIBRARY_PATH` according to `$RSFROOT/etc/madagascar/env.sh` or `$RSFROOT/etc/madagascar/env.csh`.

MATLAB INTERFACE

The MATLAB clip function is listed below.

```
1  function clip(in,out,clip)
2  %CLIP Clip the data
3
4  dims = rsf_dim(in);
5  n1 = dims(1);              % trace length
6  n2 = prod(dims(2:end));    % number of traces
7  trace = 1:n1;              % allocate trace
8  rsf_create(out,in)         % create an output file
9
10 for i2 = 1:n2              % loop over traces
11     rsf_read(trace,in,'same');
12     trace(trace >  clip) =  clip;
13     trace(trace < -clip) = -clip;
14     rsf_write(trace,out,'same');
15 end
```

Let us examine it in detail.

```
4  dims = rsf_dim(in);
```

We start by figuring out the input file dimensions.

```
5  n1 = dims(1);              % trace length
6  n2 = prod(dims(2:end));    % number of traces
```

The first dimension is the trace length, the product of all other dimensions correspond to the number of traces.

```
7  trace = 1:n1;              % allocate trace
8  rsf_create(out,in)         % create an output file
```

Next, we allocate the trace array and create an output file.

```
10 for i2 = 1:n2              % loop over traces
11     rsf_read(trace,in,'same');
12     trace(trace >  clip) =  clip;
```

```
13      trace(trace < - clip) = -clip;
14      rsf_write(trace,out,'same');
15  end
```

Finally, we do the actual work: loop over input traces, reading, clipping, and writing out each trace.

Compiling

The MATLAB script does not require compilation. Simply make sure that `$RSFROOT/lib` is in `MATLABPATH` and `LD_LIBRARY_PATH`.

INSTALLATION

To install the interface to a particular language, use `API=` parameter in the RSF configuration. For example, to to install C++ and Fortran-90 API bindings in addition to the basic package, run

```
scons API=c++,f90 config
```

Only the C interface is configured by default. The configuration parameters are stored in
`$RSFROOT/share/madagascar/etc/config.py`.

REFERENCES

Roberts, E. S., 1998, Programming abstractions in C: Addison-Wesley.