Guide to Madagascar programs

Sergey Fomel[1]

ABSTRACT

This guide introduces some of the most used `madagascar` programs and illustrates their usage with examples.

MAIN PROGRAMS

The source files for these programs can be found under system/main in the Madagascar distribution.

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

sfadd: Add, multiply, or divide RSF datasets.

```
sfadd > out.rsf scale= add= sqrt= abs= log= exp= mode= [< file0.rsf] file1.rsf
file2.rsf ...

The various operations, if selected, occur in the following order:

(1) Take absolute value, abs=
(2) Add a scalar, add=
(3) Take the natural logarithm, log=
(4) Take the square root, sqrt=
(5) Multiply by a scalar, scale=
(6) Compute the base-e exponential, exp=
(7) Add, multiply, or divide the data sets, mode=

sfadd operates on integer, float, or complex data, but all the input
and output files must be of the same data type.

An alternative to sfadd is sfmath, which is more versatile, but may be
less efficient.
```

bools	**abs=**	If true take absolute value [nin]
floats	**add=**	Scalar values to add to each dataset [nin]
bools	**exp=**	If true compute exponential [nin]
bools	**log=**	If true take logarithm [nin]
string	**mode=**	'a' means add (default), 'p' or 'm' means multiply, 'd' means divide
floats	**scale=**	Scalar values to multiply each dataset with [nin]
bools	**sqrt=**	If true take square root [nin]

`sfadd` is useful for combining (adding, dividing, or multiplying) several datasets. What if you want to subtract two datasets? Easy. Use the `scale` parameter as follows:

```
bash$ sfadd data1.rsf data2.rsf scale=1,-1 > diff.rsf
```

or

```
bash$ sfadd < data1.rsf data2.rsf scale=1,-1 > diff.rsf
```

The same task can be accomplished with the more general `sfmath` program:

```
bash$ sfmath one=data1.rsf two=data2.rsf output='one-two' > diff.rsf
```

or

```
bash$ sfmath < data1.rsf two=data2.rsf output='input-two' > diff.rsf
```

In both cases, the size and shape of `data1.rsf` and `data2.rsf` hypercubes should be the same, and a warning message is printed out if the the axis sampling parameters (such as `o1` or `d1`) in these files are different.

Implementation: system/main/add.c

The first input file is either in the list or in the standard input.

system/main/add.c

```
103     /* find number of input files */
104     if (isatty(fileno(stdin))) {
105         /* no input file in stdin */
106         nin=0;
107     } else {
108         in[0] = sf_input("in");
109         nin=1;
110     }
```

Collect input files in the `in` array from all command-line parameters that don't contain an "`=`" sign. The total number of input files is `nin`.

system/main/add.c

```
112     for (i=1; i< argc; i++) { /* collect inputs */
113         if (NULL != strchr(argv[i],'=')) continue;
114         in[nin] = sf_input(argv[i]);
115         nin++;
116     }
117     if (0==nin) sf_error ("no input");
118     /* nin = no of input files*/
```

A helper function `check_compat` checks the compatibility of input files.

Finally, we enter the main loop, where the input data are getting read buffer by buffer and combined in the total product depending on the data type.

The data combination program for floating point numbers is `add_float`.

system/main/add.c

```
424 check_compat (sf_datatype type /* data type */,
425                size_t      nin  /* number of files */,
426                sf_file*    in   /* input files [nin] */,
427                int         dim  /* file dimensionality */,
428                const off_t* n   /* dimensions [dim] */)
429 /* Check that the input files are compatible.
430    Issue error for type mismatch or size mismatch.
431    Issue warning for grid parameters mismatch. */
432 {
433     int ni, id;
434     size_t i;
435     float d, di, o, oi;
436     char key[3];
437     const float tol=1.e-5; /* tolerance for comparison */
438 
439     for (i=1; i < nin; i++) {
440         if (sf_gettype(in[i]) != type)
441             sf_error ("type mismatch: need %d",type);
442         for (id=1; id <= dim; id++) {
443             (void) snprintf(key,3,"n%d",id);
444             if (!sf_histint(in[i],key,&ni) || ni != n[id-1])
445 #if defined(__cplusplus) || defined(c_plusplus)
446                 sf_error("%s mismatch: need %ld",key,
447                          (long int) n[id-1]);
448 #else
449                 sf_error("%s mismatch: need %lld",key,
450                          (long long int) n[id-1]);
451 #endif
452             (void) snprintf(key,3,"d%d",id);
453             if (sf_histfloat(in[0],key,&d)) {
454                 if (!sf_histfloat(in[i],key,&di) ||
455                     (fabsf(di-d) > tol*fabsf(d)))
456                     sf_warning("%s mismatch: need %g",key,d);
457             } else {
458                 d = 1.;
459             }
460             (void) snprintf(key,3,"o%d",id);
461             if (sf_histfloat(in[0],key,&o) &&
462                 (!sf_histfloat(in[i],key,&oi) ||
463                  (fabsf(oi-o) > tol*fabsf(d))))
464                 sf_warning("%s mismatch: need %g",key,o);
465         }
466     }
467 }
```

system/main/add.c

```
183     for (nbuf /= sf_esize(in[0]); nsiz > 0; nsiz -= nbuf) {
184         if (nbuf > nsiz) nbuf=nsiz;
185 
186         for (j=0; j < nin; j++) {
187             collect = (bool) (j != 0);
188             switch(type) {
189                 case SF_FLOAT:
190                     sf_floatread((float*) bufi,
191                                  nbuf,
192                                  in[j]);
193                     add_float(collect,
194                               nbuf,
195                               (float*) buf,
196                               (const float*) bufi,
197                               cmode,
198                               scale[j],
199                               add[j],
200                               abs_flag[j],
201                               log_flag[j],
202                               sqrt_flag[j],
203                               exp_flag[j]);
```

system/main/add.c

```
264 static void add_float (bool   collect,    /* if collect */
265                         size_t nbuf,       /* buffer size */
266                         float* buf,        /* output [nbuf] */
267                         const float* bufi, /* input  [nbuf] */
268                         char   cmode,      /* operation */
269                         float  scale,      /* scale factor */
270                         float  add,        /* add factor */
271                         bool   abs_flag,   /* if abs */
272                         bool   log_flag,   /* if log */
273                         bool   sqrt_flag,  /* if sqrt */
274                         bool   exp_flag    /* if exp */)
275 /* Add floating point numbers */
276 {
277     size_t j;
278     float f;
279 
280     for (j=0; j < nbuf; j++) {
281         f = bufi[j];
282         if (abs_flag)    f = fabsf(f);
283         f += add;
284         if (log_flag)    f = logf(f);
285         if (sqrt_flag)   f = sqrtf(f);
286         if (1. != scale) f *= scale;
287         if (exp_flag)    f = expf(f);
288         if (collect) {
289             switch (cmode) {
290                 case 'p': /* product */
291                 case 'm': /* multiply */
292                     buf[j] *= f;
293                     break;
294                 case 'd': /* delete */
295                     if (f != 0.) buf[j] /= f;
296                     break;
297                 default:  /* add */
298                     buf[j] += f;
299                     break;
300             }
301         } else {
302             buf[j] = f;
303         }
304     }
305 }
```

sfattr: Display dataset attributes.

```
sfattr < in.rsf lval=2 want=
```

```
Sample output from "sfspike n1=100 | sfbandpass fhi=60 | sfattr"
****************************************
     rms =      0.992354
    mean =      0.987576
  2-norm =       9.92354
variance =    0.00955481
 std dev =     0.0977487
     max =       1.12735 at 97
     min =      0.151392 at 100
nonzero samples = 100
  total samples = 100
****************************************
```

```
rms                 = sqrt[ sum(data^2) / n ]
mean                = sum(data) / n
norm                = sum(abs(data)^lval)^(1/lval)
variance            = [ sum(data^2) - n*mean^2 ] / [ n-1 ]
standard deviation = sqrt [ variance ]
```

int	**lval=2**	norm option, lval is a non-negative integer, computes the vector lval-norm
string	**want=**	'all'(default), 'rms', 'mean', 'norm', 'var', 'std', 'max', 'min', 'nonzero', 'samples', 'short' want= 'rms' displays the root mean square want= 'norm' displays the square norm, otherwise specified by lval. want= 'var' displays the variance want= 'std' displays the standard deviation want= 'nonzero' displays number of nonzero samples want= 'samples' displays total number of samples want= 'short' displays a short one-line version

`sfattr` is a useful diagnostic program. It reports certain statistical values for an RSF dataset: RMS (root-mean-square) amplitude, mean value, norm value, variance, standard deviation, maximum and minimum values, number of nonzero samples, and the total number of samples.

If we denote data values as d_i for $i = 0, 1, 2, \ldots, n$, then the RMS value is $\sqrt{\frac{1}{n}\sum_{i=0}^{n} d_i^2}$, the mean value is $\frac{1}{n}\sum_{i=0}^{n} d_i$, the L_2-norm value is $\sqrt{\sum_{i=0}^{n} d_i^2}$, the variance is $\frac{1}{n-1}\left[\sum_{i=0}^{n} d_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n} d_i\right)^2\right]$, and the standard deviation is the square root of the variance. Using `sfattr` is a quick way to see the distribution of data values and check it for anomalies.

Implementation: system/main/attr.c

Computations start by finding the input data (`in`) size (`nsiz`) and dimensions (`dim`).

system/main/attr.c

```
81      dim = (size_t) sf_largefiledims (in,n);
82      for (nsiz=1, i=0; i < dim; i++) {
83          nsiz *= n[i];
84      }
```

In the main loop, we read the input data buffer by buffer.

system/main/attr.c

```
100     for (nleft=nsiz; nleft > 0; nleft -= nbuf) {
101         nbuf = (bufsiz < nleft)? bufsiz: nleft;
102         switch (type) {
103             case SF_FLOAT:
104                 sf_floatread((float*) buf,nbuf,in);
105                 break;
106             case SF_INT:
107                 sf_intread((int*) buf,nbuf,in);
108                 break;
109             case SF_SHORT:
110                 sf_shortread((short*) buf,nbuf,in);
111                 break;
112             case SF_COMPLEX:
113                 sf_complexread((sf_complex*) buf,nbuf,in);
114                 break;
115             case SF_UCHAR:
116                 sf_ucharread((unsigned char*) buf,nbuf,in);
117                 break;
118             case SF_CHAR:
119             default:
120                 sf_charread(buf,nbuf,in);
121                 break;
122         }
```

The data attributes are accumulated in corresponding double-precision variables.

Finally, the attributes are reduced and printed out.

system/main/attr.c

```
146                 fsum += f;
147                 fsqr += (double) f*f;
```

system/main/attr.c

```
180     fmean = fsum/nsiz;
181     if (lval==2)       fnorm = sqrt(fsqr);
182     else if (lval==0) fnorm = nsiz-nzero;
183     else               fnorm = pow(flval,1./lval);
184     frms = sqrt(fsqr/nsiz);
185     if (nsiz > 1) fvar = fabs(fsqr-nsiz*fmean*fmean)/(nsiz-1);
186     else          fvar = 0.0;
187     fstd = sqrt(fvar);
```

system/main/attr.c

```
194     if(NULL==want || 0==strcmp(want,"rms"))
195         printf("      rms = %13.6g \n",(float) frms);
196     if(NULL==want || 0==strcmp(want,"mean"))
197         printf("     mean = %13.6g \n",(float) fmean);
198     if(NULL==want || 0==strcmp(want,"norm"))
199         printf("  %d-norm = %13.6g \n",lval,(float) fnorm);
200     if(NULL==want || 0==strcmp(want,"var"))
201         printf("variance = %13.6g \n",(float) fvar);
202     if(NULL==want || 0==strcmp(want,"std"))
203         printf(" std dev = %13.6g \n",(float) fstd);
```

sfcat: Concatenate datasets.

```
sfcat > out.rsf order= space= axis=3 nspace=(int) (ni/(20*nin) + 1) o= d=
[<file0.rsf] file1.rsf file2.rsf ...

sfmerge inserts additional space between merged data.
```

int	**axis=3**		Axis being merged
float	**d=**		axis sampling
int	**nspace=(int) (ni/(20*nin) + 1)**		if space=y, number of traces to insert
float	**o=**		axis origin
ints	**order=**		concatenation order [nin]
bool	**space=**	[y/n]	Insert additional space. y is default for sfmerge, n is default for sfcat

`sfcat` and `sfmerge` concatenate two or more files together along a particular axis. It is the same program, only `sfcat` has the default `space=n` and `sfmerge` has the default `space=y`.

Example of `sfcat`:

```
bash$ sfspike n1=2 n2=3 > one.rsf
bash$ sfin one.rsf
one.rsf:
    in="/tmp/one.rsf@"
    esize=4 type=float form=native
    n1=2            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
        6 elements 24 bytes
bash$ sfcat one.rsf one.rsf axis=1 > two.rsf
bash$ sfin two.rsf
two.rsf:
    in="/tmp/two.rsf@"
    esize=4 type=float form=native
    n1=4            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
        12 elements 48 bytes
```

Example of `sfmerge`:

```
bash$ sfmerge one.rsf one.rsf axis=2 > two.rsf
bash$ sfin two.rsf
two.rsf:
    in="/tmp/two.rsf@"
```

```
    esize=4 type=float form=native
    n1=2            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=7            d2=0.1          o2=0            label2="Distance" unit2="km"
        14 elements 56 bytes
```

In this case, an extra empty trace is inserted between the two merged files.

The axes that are not being merged are checked for consistency:

```
bash$ sfcat one.rsf two.rsf > three.rsf
sfcat: n2 mismatch: need 3
```

Implementation: system/main/cat.c

The first input file is either in the list or in the standard input.

system/main/cat.c

```
64      if (!sf_stdin()) { /* no input file in stdin */
65          nin=0;
66      } else {
67          filename[0] = "in";
68          nin=1;
69      }
```

Everything on the command line that does not contain a "=" sign is treated as a file name, and the corresponding file object is added to the list.

system/main/cat.c

```
71      for (i=1; i< argc; i++) { /* collect inputs */
72          if (NULL != strchr(argv[i],'='))
73              continue; /* not a file */
74          filename[nin] = argv[i];
75          nin++;
76      }
77      if (0==nin) sf_error ("no input");
```

As explained above, if the `space=` parameter is not set, it is inferred from the program name: `sfmerge` corresponds to `space=y` and `sfcat` corresponds to `space=n`.

Find the axis for the merging (from the command line `axis=` argument) and figure out two sizes: `n1` for everything after the axis and `n2` for everything before the axis.

system/main/cat.c

```
99      if (!sf_getbool("space",&space)) {
100         /* Insert additional space.
101            y is default for sfmerge, n is default for sfcat */
102         prog = sf_getprog();
103         if (NULL != strstr (prog, "merge")) {
104             space = true;
105         } else if (NULL != strstr (prog, "cat")) {
106             space = false;
107         } else {
108             sf_warning("%s is neither merge nor cat,"
109                        " assume merge",prog);
110             space = true;
111         }
112     }
```

system/main/cat.c

```
132     n1=1;
133     n2=1;
134     for (i=1; i <= dim; i++) {
135         if      (i < axis) n1 *= n[i-1];
136         else if (i > axis) n2 *= n[i-1];
137     }
```

In the output, the selected axis will get extended.

system/main/cat.c

```
149     /* figure out the length of extended axis */
150     ni = 0;
151     for (j=0; j < nin; j++) {
152         ni += naxis[j];
153     }
154 
155     if (space) {
156         if (!sf_getint("nspace",&nspace))
157             nspace = (int) (ni/(20*nin) + 1);
158         /* if space=y, number of traces to insert */
159         ni += nspace*(nin-1);
160     }
161 
162     (void) snprintf(key,3,"n%d",axis);
163     sf_putint(out,key,(int) ni);
```

The rest is simple: loop through the datasets reading and writing the data in buffer-size chunks and adding extra empty chunks if `space=y`.

sfcmplx: Create a complex dataset from its real and imaginary parts.

```
sfcmplx < real.rsf > cmplx.rsf real.rsf imag.rsf

There has to be only two input files specified and no additional parameters.
```

`sfcmplx` simply creates a complex dataset from its real and imaginary parts. The reverse operation can be accomplished with `sfreal` and `sfimag`.

Example of `sfcmplx`:

```
bash$ sfspike n1=2 n2=3 > one.rsf
bash$ sfin one.rsf
one.rsf:
    in="/tmp/one.rsf@"
    esize=4 type=float form=native
    n1=2            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
        6 elements 24 bytes
```

system/main/cat.c

```
184     for (i2=0; i2 < n2; i2++) {
185         for (j=0; j < nin; j++) {
186             k = order[j];
187             for (ni = n1*naxis[k]*esize; ni > 0; ni -= nbuf) {
188                 nbuf = (BUFSIZ < ni)? BUFSIZ: ni;
189                 sf_charread (buf,nbuf,in[k]);
190                 sf_charwrite (buf,nbuf,out);
191             }
192             if (!space || j == nin-1) continue;
193             /* Add spaces */
194             memset(buf,0,BUFSIZ);
195             for (ni = n1*nspace*esize; ni > 0; ni -= nbuf) {
196                 nbuf = (BUFSIZ < ni)? BUFSIZ: ni;
197                 sf_charwrite (buf,nbuf,out);
198             }
199         }
200     }
```

```
bash$ sfcmplx one.rsf one.rsf > cmplx.rsf
bash$ sfin cmplx.rsf
cmplx.rsf:
    in="/tmp/cmplx.rsf@"
    esize=8 type=complex form=native
    n1=2            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
        6 elements 48 bytes
```

Implementation: system/main/cmplx.c

The program flow is simple. First, get the names of the input files.

The main part of the program reads the real and imaginary parts buffer by buffer and assembles and writes out the complex input.

sfconjgrad: Generic conjugate-gradient solver for linear inversion

```
sfconjgrad < dat.rsf mod=mod.rsf > to.rsf < from.rsf > out.rsf niter=1
```

file	**mod=**	auxiliary input file name
int	**niter=1**	number of iterations

system/main/cmplx.c

```
41      /* the first two non-parameters are real and imag files */
42      for (i=1; i< argc; i++) {
43          if (NULL == strchr(argv[i],'=')) {
44              if (NULL == real) {
45                  real = sf_input (argv[i]);
46              } else {
47                  imag = sf_input (argv[i]);
48                  break;
49              }
50          }
51      }
52      if (NULL == imag) {
53          if (NULL == real) sf_error ("not enough input");
54          /* if only one input, real is in stdin */
55          imag = real;
56          real = sf_input("in");
57      }
```

system/main/cmplx.c

```
81      for (nleft= (size_t) (rsize*resize);
82           nleft > 0; nleft -= nbuf) {
83          nbuf = (BUFSIZ < nleft)? BUFSIZ: nleft;
84          sf_charread(rbuf,nbuf,real);
85          sf_charread(ibuf,nbuf,imag);
86          for (i=0; i < nbuf; i += resize) {
87              memcpy(cbuf+2*i,        rbuf+i,(size_t) resize);
88              memcpy(cbuf+2*i+resize,ibuf+i,(size_t) resize);
89          }
90          sf_charwrite(cbuf,2*nbuf,cmplx);
91      }
```

`sfconjgrad` is a generic program for least-squares linear inversion with the conjugate-gradient method. Suppose you have an executable program `<prog>` that takes an RSF file from the standard input and produces an RSF file in the standard output. It may take any number of additional parameters but one of them must be `adj=` that sets the forward (`adj=0`) or adjoint (`adj=1`) operations. The program `<prog>` is typically an RSF program but it could be anything (a script, a multiprocessor MPI program, etc.) as long as it implements a linear operator $\mathbf{L}$ and its adjoint. There are no restrictions on the data size or shape. You can easily test the adjointness with `sfdottest`. The `sfconjgrad` program searches for a vector $\mathbf{m}$ that minimizes the least-square misfit $\|\mathbf{d} - \mathbf{L}\,\mathbf{m}\|^2$ for the given input data vector $\mathbf{d}$.

Here is an example. The `sfhelicon` program implements Claerbout's multidimensional helical filtering (Claerbout, 1998). It requires a filter to be specified in addition to the input and output vectors. We create a helical 2-D filter using the Unix `echo` command.

```
bash$ echo 1 19 20 n1=3 n=20,20 data_format=ascii_int in=lag.rsf > lag.rsf
bash$ echo 1 1 1 a0=-3 n1=3 data_format=ascii_float in=flt.rsf > flt.rsf
```

Next, we create an example 2-D model and data vector with `sfspike`.

```
bash$ sfspike n1=50 n2=50 > vec.rsf
```

The `sfdottest` program can perform the dot product test to check that the adjoint mode works correctly.

```
bash$ sfdottest sfhelicon filt=flt.rsf lag=lag.rsf \
mod=vec.rsf dat=vec.rsf
sfdottest:  L[m]*d=5.28394
sfdottest: L'[d]*m=5.28394
```

Your numbers may be different because `sfdottest` generates new random input on each run. Next, let us make some random data with `sfnoise`.

```
bash$ sfnoise seed=2005 rep=y < vec.rsf > dat.rsf
```

and try to invert the filtering operation using `sfconjgrad`:

```
bash$ sfconjgrad sfhelicon filt=flt.rsf lag=lag.rsf \
mod=vec.rsf < dat.rsf > mod.rsf niter=10
sfconjgrad: iter 1 of 10
sfconjgrad: grad=3253.65
sfconjgrad: iter 2 of 10
sfconjgrad: grad=289.421
```

```
sfconjgrad: iter 3 of 10
sfconjgrad: grad=92.3481
sfconjgrad: iter 4 of 10
sfconjgrad: grad=36.9417
sfconjgrad: iter 5 of 10
sfconjgrad: grad=18.7228
sfconjgrad: iter 6 of 10
sfconjgrad: grad=11.1794
sfconjgrad: iter 7 of 10
sfconjgrad: grad=7.26941
sfconjgrad: iter 8 of 10
sfconjgrad: grad=5.15945
sfconjgrad: iter 9 of 10
sfconjgrad: grad=4.23055
sfconjgrad: iter 10 of 10
sfconjgrad: grad=3.57495
```

The output shows that, in 10 iterations, the norm of the gradient vector decreases by almost 1000. We can check the residual misfit before

```
bash$ < dat.rsf sfattr want=norm
norm value = 49.7801
```

and after

```
bash$ sfhelicon filt=flt.rsf lag=lag.rsf < mod.rsf | \
sfadd scale=1,-1 dat.rsf | sfattr want=norm
norm value = 5.73563
```

In 10 iterations, the misfit decreased by an order of magnitude. The result can be improved by running the program for more iterations.

Implementation: system/main/conjgrad.c

sfcp: Copy or move a dataset.

```
sfcp < in.rsf > out.rsf in.rsf out.rsf

sfcp - copy, sfmv - move.
Mimics standard Unix commands.
```

The `sfcp` and `sfmv` command imitate the Unix `cp` and `mv` commands and serve for copying and moving RSF files. Example:

```
bash$ sfspike n1=2 n2=3 > one.rsf
bash$ sfin one.rsf
one.rsf:
    in="/tmp/one.rsf@"
    esize=4 type=float form=native
    n1=2            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
        6 elements 24 bytes
bash$ sfcp one.rsf two.rsf
bash$ sfin two.rsf
two.rsf:
    in="/tmp/two.rsf@"
    esize=4 type=float form=native
    n1=2            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
        6 elements 24 bytes
```

Implementation: system/main/cp.c

First, we look for the two first command-line arguments that don't have the "=" character in them and consider them as the names of the input and the output files.

system/main/cp.c

```
47          /* the first two non-parameters are in and out files */
48          for (i=1; i< argc; i++) {
49              if (NULL == strchr(argv[i],'=')) {
50                  if (NULL == in) {
51                      infile = argv[i];
52                      in = sf_input (infile);
53                  } else {
54                      out = sf_output (argv[i]);
55                      break;
56                  }
57              }
58          }
```

Next, we use library functions `sf_cp` and `sf_mv` to do the actual work.

system/main/cp.c

```
66      sf_cp(in,out);
67      if (NULL != strstr (prog,"mv"))
68          sf_rm(infile,false,false,false);
```

sfcut: Zero a portion of the dataset.

```
sfcut < in.rsf > out.rsf verb=n j#=(1,...)  d#=(d1,d2,...)  f#=(0,...)
min#=(o1,o2,,...)  n#=(0,...)  max#=(o1+(n1-1)*d1,o2+(n1-1)*d2,,...)

Reverse of window.
```

float	**d#=(d1,d2,...)**		sampling in #-th dimension
largeint	**f#=(0,...)**		window start in #-th dimension
int	**j#=(1,...)**		jump in #-th dimension
float	**max#=(o1+(n1-1)*d1,o2+(n1-1)*d2,,...)**		maximum in #-th dimension
float	**min#=(o1,o2,,...)**		minimum in #-th dimension
int	**n#=(0,...)**		window size in #-th dimension
bool	**verb=n**	[y/n]	Verbosity flag

The `sfcut` command is related to `sfwindow` and has the same set of arguments only instead of extracting the selected window, it fills it with zeroes. The size of the input data is preserved.

Examples:

```
bash$ sfspike n1=5 n2=5 > in.rsf
bash$ < in.rsf sfdisfil
   0:             1             1             1             1             1
   5:             1             1             1             1             1
  10:             1             1             1             1             1
  15:             1             1             1             1             1
  20:             1             1             1             1             1
bash$ < in.rsf sfcut n1=2 f1=1 n2=3 f2=2 | sfdisfil
   0:             1             1             1             1             1
   5:             1             1             1             1             1
  10:             1             0             0             1             1
  15:             1             0             0             1             1
  20:             1             0             0             1             1
bash$ < in.rsf sfcut j1=2 | sfdisfil
   0:             0             1             0             1             0
   5:             0             1             0             1             0
```

```
  10:           0           1           0           1           0
  15:           0           1           0           1           0
  20:           0           1           0           1           0
```

sfdd: Convert between different formats.

```
sfdd < in.rsf > out.rsf trunc=n line=8 ibm=n form= type= format=
```

string	**form=**		ascii, native, xdr
string	**format=**		Element format (for conversion to ASCII)
bool	**ibm=n**	[y/n]	Special case - assume integers actually represent IBM floats
int	**line=8**		Number of numbers per line (for conversion to ASCII)
bool	**trunc=n**	[y/n]	Truncate or round to nearest when converting from float to int/short
string	**type=**		int, float, complex, short

The `sfdd` program is used to change either the form (`ascii`, `xdr`, `native`) or the type (`complex`, `float`, `int`, `char`) of the input dataset.

In the example below, we create a plain text (ASCII) file with numbers and then use `sfdd` to generate an RSF file in `xdr` form with `complex` numbers.

```
bash$ cat test.txt
1 2 3 4 5 6
bash$ echo n1=6 data_format=ascii_int in=test.txt > test.rsf
bash$ sfin test.rsf
test.rsf:
    in="test.txt"
    esize=0 type=int form=ascii
    n1=6            d1=?            o1=?
        6 elements
bash$ sfdd < test.rsf form=xdr type=complex > test2.rsf
bash$ sfin test2.rsf
test2.rsf:
    in="/tmp/test2.rsf@"
    esize=8 type=complex form=xdr
    n1=3            d1=?            o1=?
        3 elements 24 bytes
bash$ sfdisfil < test2.rsf
   0:          1,         2i          3,         4i          5,         6i
```

To learn more about the RSF data format, consult the guide to RSF format.

sfdisfil: Print out data values.

```
sfdisfil < in.rsf number=y col=0 format= header= trailer=

Alternatively, use sfdd and convert to ASCII form.
```

int	**col=0**		Number of columns. The default depends on the data type: 10 for int and char, 5 for float, 3 for complex
string	**format=**		Format for numbers (printf-style). The default depends on the data type: """
string	**header=**		Optional header string to output before data
bool	**number=y**	[y/n]	If number the elements
string	**trailer=**		Optional trailer string to output after data

The `sfdisfil` program simply dumps the data contents to the standard output in a text form. It is used mostly for debugging purposes to quickly examine RSF files. Here is an example:

```
bash$ sfmath o1=0 d1=2 n1=12 output=x1 > test.rsf
bash$ < test.rsf sfdisfil
   0:              0             2             4             6             8
   5:             10            12            14            16            18
  10:             20            22
```

The output format is easily configurable.

```
bash$ < test.rsf sfdisfil col=6 number=n format="%5.1f"
  0.0  2.0  4.0  6.0  8.0 10.0
 12.0 14.0 16.0 18.0 20.0 22.0
```

Along with `sfdd`, `sfdisfil` provides a simple way to convert RSF data to an ASCII form.

sfdottest: Generic dot-product test for linear operators with adjoints

```
sfdottest mod=mod.rsf dat=dat.rsf > pip.rsf
```

file	**dat=**	auxiliary input file name
file	**mod=**	auxiliary input file name

`sfdottest` is a generic dot-product test program for testing linear operators. Suppose there is an executable program `<prog>` that takes an RSF file from the standard

input and produces an RSF file in the standard output. It may take any number of additional parameters but one of them must be `adj=` that sets the forward (`adj=0`) or adjoint (`adj=1`) operations. The program `<prog>` is typically an RSF program but it could be anything (a script, a multiprocessor MPI program, etc.) as long as it implements a linear operator $\mathbf{L}$ and its adjoint $\mathbf{L}^T$. The `sfdottest` program is testing the equality

$$\mathbf{d}^T \mathbf{L}\,\mathbf{m} = \mathbf{m}^T \mathbf{L}^T \mathbf{d} \tag{1}$$

by using random vectors $\mathbf{m}$ and $\mathbf{d}$. You can invoke it with

```
bash$ sfdottest <prog> [optional aruments] mod=mod.rsf dat=dat.rsf
```

where `mod.rsf` and `dat.rsf` are RSF files that represent vectors from the model and data spaces. `sfdottest` does not create any temporary files and does not have any restrictive limitations on the size of the vectors.

Here is an example. We first setup a vector with 100 elements using `sfspike` and then run `sfdottest` to test the `sfcausint` program. `sfcausint` implements a linear operator of causal integration and its adjoint, the anti-causal integration.

```
bash$ sfspike n1=100 > vec.rsf
bash$ sfdottest sfcausint mod=vec.rsf dat=vec.rsf
sfdottest:  L[m]*d=1410.2
sfdottest: L'[d]*m=1410.2
bash$ sfdottest sfcausint mod=vec.rsf dat=vec.rsf
sfdottest:  L[m]*d=1165.87
sfdottest: L'[d]*m=1165.87
```

The numbers are different on subsequent runs because of changing seed in the random number generator.

Here is a somewhat more complicated example. The `sfhelicon` program implements Claerbout's multidimensional helical filtering (Claerbout, 1998). It requires a filter to be specified in addition to the input and output vectors. We create a helical 2-D filter using the Unix `echo` command.

```
bash$ echo 1 19 20 n1=3 n=20,20 data_format=ascii_int in=lag.rsf > lag.rsf
bash$ echo 1 1 1 a0=-3 n1=3 data_format=ascii_float in=flt.rsf > flt.rsf
```

Next, we create an example 2-D model and data vector with `sfspike`.

```
bash$ sfspike n1=50 n2=50 > vec.rsf
```

Now the `sfdottest` program can perform the dot product test.

```
bash$ sfdottest sfhelicon filt=flt.rsf lag=lag.rsf \
> mod=vec.rsf dat=vec.rsf
sfdottest:  L[m]*d=8.97375
sfdottest: L'[d]*m=8.97375
```

Here is the same program tested in the inverse filtering mode:

```
bash$ sfdottest sfhelicon filt=flt.rsf lag=lag.rsf \
> mod=vec.rsf dat=vec.rsf div=y
sfdottest:  L[m]*d=15.0222
sfdottest: L'[d]*m=15.0222
```

sfget: Output parameters from the header.

```
sfget parform=y all=n par1 par2 ...
```

bool	**all=n**	[y/n]	If output all values.
bool	**parform=y**	[y/n]	If y, print out parameter=value. If n, print out value.

The `sfget` program extracts a parameter value from an RSF file. It is useful mostly for scripting. Here is, for example, a quick calculation of the maximum value on the first axis in an RSF dataset (the output of `sfspike`) using the standard Unix `bc` calculator.

```
bash$ ( sfspike n1=100 | sfget n1 d1 o1; echo "o1+(n1-1)*d1" ) | bc
.396
```

See also `sfput`.

Implementation: system/main/get.c

The implementation is trivial. Loop through all command-line parameters that contain the "=" character.

system/main/get.c

```
41  if(!sf_getbool("all",&all)) all=false;
42  /* If output all values. */
```

Get the parameter value (as string) and output it as either `key=value` or `value`, depending on the `parform` parameter.

system/main/get.c

```
44      table = sf_simtab_init(tabsize);
45      sf_simtab_input(table, stdin, NULL);
46
47      if (all) {
48          sf_simtab_output(table, stdout);
49      } else {
50          for (i = 1; i < argc; i++) {
```

sfheadercut: Zero a portion of a dataset based on a header mask.

```
sfheadercut mask=head.rsf < in.rsf > out.rsf

The input data is a collection of traces n1xn2,
mask is an integer array of size n2.
```

file	**mask=**	auxiliary input file name

`sfheadercut` is close to `sfheaderwindow` but instead of windowing the dataset, it fills the traces specified by the header mask with zeroes. The size of the input data is preserved.

Here is an example of using `sfheaderwindow` for zeroing every other trace in the input file. First, let us create an input file with ten traces:

```
bash$ sfmath n1=5 n2=10 output=x2+1 > input.rsf
bash$ < input.rsf sfdisfil
   0:             1             1             1             1             1
   5:             2             2             2             2             2
  10:             3             3             3             3             3
  15:             4             4             4             4             4
  20:             5             5             5             5             5
  25:             6             6             6             6             6
  30:             7             7             7             7             7
  35:             8             8             8             8             8
  40:             9             9             9             9             9
  45:            10            10            10            10            10
```

Next, we can create a mask with alternating ones and zeros using `sfinterleave`.

```
bash$ sfspike n1=5 mag=1 | sfdd type=int > ones.rsf
```

```
bash$ sfspike n1=5 mag=0 | sfdd type=int > zeros.rsf
bash$ sfinterleave axis=1 ones.rsf zeros.rsf > mask.rsf
bash$ sfdisfil < mask.rsf
   0:    1    0    1    0    1    0    1    0    1    0
```

Finally, `sfheadercut` zeros the input traces.

```
bash$ sfheadercut < input.rsf mask=mask.rsf > output.rsf
bash$ sfdisfil < output.rsf
   0:            1            1            1            1            1
   5:            0            0            0            0            0
  10:            3            3            3            3            3
  15:            0            0            0            0            0
  20:            5            5            5            5            5
  25:            0            0            0            0            0
  30:            7            7            7            7            7
  35:            0            0            0            0            0
  40:            9            9            9            9            9
  45:            0            0            0            0            0
```

sfheadersort: Sort a dataset according to a header key.

```
sfheadersort < in.rsf > out.rsf head=
```

string	**head=**	header file

`sfheadersort` is used to sort traces in the input file according to trace header information.

Here is an example of using `sfheadersort` for randomly shuffling traces in the input file. First, let us create an input file with seven traces:

```
bash$ sfmath n1=5 n2=7 output=x2+1 > input.rsf
bash$ < input.rsf sfdisfil
   0:            1            1            1            1            1
   5:            2            2            2            2            2
  10:            3            3            3            3            3
  15:            4            4            4            4            4
  20:            5            5            5            5            5
  25:            6            6            6            6            6
  30:            7            7            7            7            7
```

Next, we can create a random file with seven header values using `sfnoise`.

```
bash$ sfspike n1=7 | sfnoise rep=y type=n > random.rsf
bash$ < random.rsf sfdisfil
   0:       0.05256      -0.2879       0.1487       0.4097       0.1548
   5:        0.4501       0.2836
```

If you reproduce this example, your numbers will most likely be different, because, in the absence of `seed=` parameter, `sfnoise` uses a random seed value to generate pseudo-random numbers. Finally, we apply `sfheadersort` to shuffle the input traces.

```
bash$ < input.rsf sfheadersort head=random.rsf > output.rsf
bash$ < output.rsf sfdisfil
   0:            2            2            2            2            2
   5:            1            1            1            1            1
  10:            3            3            3            3            3
  15:            5            5            5            5            5
  20:            7            7            7            7            7
  25:            4            4            4            4            4
  30:            6            6            6            6            6
```

As expected, the order of traces in the output file corresponds to the order of values in the header. Thanks to the separation between headers and data, the operation of `sfheadersort` is optimally efficient. It first sorts the headers and only then accesses the data, reading each data trace only once.

sfheaderwindow: Window a dataset based on a header mask.

```
sfheaderwindow mask=head.rsf < in.rsf > out.rsf

The input data is a collection of traces n1xn2,
mask is an integer array os size n2, windowed is n1xm2,
where m2 is the number of nonzero elements in mask.
```

file	**mask=**	auxiliary input file name

`sfheaderwindow` is used to window traces in the input file according to trace header information.

Here is an example of using `sfheaderwindow` for randomly selecting part of the traces in the input file. First, let us create an input file with ten traces:

```
bash$ sfmath n1=5 n2=10 output=x2+1 > input.rsf
bash$ < input.rsf sfdisfil
   0:            1            1            1            1            1
   5:            2            2            2            2            2
  10:            3            3            3            3            3
```

```
 15:           4           4           4           4           4
 20:           5           5           5           5           5
 25:           6           6           6           6           6
 30:           7           7           7           7           7
 35:           8           8           8           8           8
 40:           9           9           9           9           9
 45:          10          10          10          10          10
```

Next, we can create a random file with ten header values using `sfnoise`.

```
bash$ sfspike n1=10 | sfnoise rep=y type=n > random.rsf
bash$ < random.rsf sfdisfil
   0:     -0.005768      0.02258     -0.04331      -0.4129      -0.3909
   5:      -0.03582       0.4595      -0.3326        0.498      -0.3517
```

If you reproduce this example, your numbers will most likely be different, because, in the absence of `seed=` parameter, `sfnoise` uses a random seed value to generate pseudo-random numbers. Finally, we apply `sfheaderwindow` to window the input traces selecting only those for which the header is greater than zero.

```
bash$ < random.rsf sfmask min=0 > mask.rsf
bash$ < mask.rsf sfdisfil
   0:    0    1    0    0    0    0    1    0    1    0
bash$ < input.rsf sfheaderwindow mask=mask.rsf > output.rsf
bash$ < output.rsf sfdisfil
   0:           2           2           2           2           2
   5:           7           7           7           7           7
  10:           9           9           9           9           9
```

In this case, only three traces are selected for the output. Thanks to the separation between headers and data, the operation of `sfheaderwindow` is optimally efficient.

sfin: Display basic information about RSF files.

```
sfin info=y check=2.  trail=y [<file0.rsf] file1.rsf file2.rsf ...

n1,n2,... are data dimensions
o1,o2,... are axis origins
d1,d2,... are axis sampling intervals
label1,label2,... are axis labels
unit1,unit2,... are axis units
```

float	**check=2.**		Portion of the data (in Mb) to check for zero values.
bool	**info=y**	[y/n]	If n, only display the name of the data file.
bool	**trail=y**	[y/n]	If n, skip trailing dimensions of one

`sfin` is one of the most useful programs for operating with RSF files. It produces quick information on the file hypercube dimensions and checks the consistency of the associated data file.

Here is an example. Let us create an RSF file and examine it with `sfin`.

```
bash$ sfspike n1=100 n2=20 > spike.rsf
bash$ sfin spike.rsf
spike.rsf:
    in="/tmp/spike.rsf@"
    esize=4 type=float form=native
    n1=100          d1=0.004        o1=0          label1="Time" unit1="s"
    n2=20           d2=0.1          o2=0          label2="Distance" unit2="km"
        2000 elements 8000 bytes
```

`sfin` reports the following information:

- location of the data file (`/tmp/spike.rsf`)
- element size (4 bytes)
- element type (floating point)
- element form (native)
- hypercube dimensions (100 by 20)
- axes scale (0.004 and 0.1)
- axes origin (0 and 0)
- axes labels
- axes units
- total number of elements
- total number of bytes in the data file

Suppose that the file got corrupted by a buggy program and reports incorrect dimensions. The `sfin` program should be able to catch the discrepancy.

```
bash$ echo n2=100 >> spike.rsf
bash$ sfin spike.rsf > /dev/null
sfin:           Actually 8000 bytes, 20% of expected.
```

`sfin` also checks the first records in the file for zeros.

```
bash$ sfspike n1=100 n2=100 k2=99 > spike2.rsf
bash$ sfin spike2.rsf >/dev/null
sfin: The first 32768 bytes are all zeros
```

The number of bytes to check is adjustable

```
bash$ sfin spike2.rsf check=0.01 >/dev/null
sfin: The first 16384 bytes are all zeros
```

You can also output only the location of the data file. This is sometimes handy in scripts.

```
bash$ sfin spike.rsf spike2.rsf info=n
/tmp/spike.rsf@ /tmp/spike2.rsf@
```

An alternative is to use `sfget`, as follows:

```
bash$ sfget parform=n in < spike.rsf
/tmp/spike.rsf@
```

sfinterleave: Combine several datasets by interleaving.

```
sfinterleave > out.rsf axis=3 [< file0.rsf] file1.rsf file2.rsf ...
```

int	**axis=3**	Axis for interleaving

`sfinterleave` combines two or more datasets by interleaving them on one of the axes. Here is a quick example:

```
bash$ sfspike n1=5 n2=5 > one.rsf
bash$ sfdisfil < one.rsf
   0:             1            1            1            1            1
   5:             1            1            1            1            1
  10:             1            1            1            1            1
  15:             1            1            1            1            1
  20:             1            1            1            1            1
bash$ sfscale < one.rsf dscale=2 > two.rsf
bash$ sfdisfil < two.rsf
   0:             2            2            2            2            2
   5:             2            2            2            2            2
  10:             2            2            2            2            2
  15:             2            2            2            2            2
  20:             2            2            2            2            2
```

```
bash$ sfinterleave one.rsf two.rsf axis=1 | sfdisfil
   0:             1            2            1            2            1
   5:             2            1            2            1            2
  10:             1            2            1            2            1
  15:             2            1            2            1            2
  20:             1            2            1            2            1
  25:             2            1            2            1            2
  30:             1            2            1            2            1
  35:             2            1            2            1            2
  40:             1            2            1            2            1
  45:             2            1            2            1            2
bash$ sfinterleave < one.rsf two.rsf axis=2 | sfdisfil
   0:             1            1            1            1            1
   5:             2            2            2            2            2
  10:             1            1            1            1            1
  15:             2            2            2            2            2
  20:             1            1            1            1            1
  25:             2            2            2            2            2
  30:             1            1            1            1            1
  35:             2            2            2            2            2
  40:             1            1            1            1            1
  45:             2            2            2            2            2
```

sfmask: Create a mask.

```
sfmask < in.rsf > out.rsf min= max= min= max=

Mask is an integer data with ones and zeros.
Ones correspond to input values between min and max.

The output can be used with sfheaderwindow.
```

int	**max=**	maximum header value
int	**min=**	minimum header value

`sfmask` creates an integer output of ones and zeros comparing the values of the input data to specified `min=` and `max=` parameters. It is useful for `sfheaderwindow` and in many other applications. Here is a quick example:

```
bash$ sfmath n1=10 output="sin(x1)" > sin.rsf
bash$ < sin.rsf sfdisfil
   0:             0       0.8415       0.9093       0.1411      -0.7568
   5:       -0.9589      -0.2794        0.657       0.9894       0.4121
bash$ < sin.rsf sfmask min=-0.5 max=0.5 | sfdisfil
   0:    1    0    0    1    0    0    1    0    0    1
```

sfmath: Mathematical operations on data files.

```
sfmath > out.rsf n#= d#=(1,1,...)  o#=(0,0,...)  label#= unit#= type= label= unit=
output=

Known functions:
cos,  sin,  tan,  acos,  asin,  atan,
cosh, sinh, tanh, acosh, asinh, atanh,
exp,  log,  sqrt, abs,
erf,  erfc (for float data),
arg,  conj, real, imag (for complex data).

sfmath will work on float or complex data, but all the input and output
files must be of the same data type.

An alternative to sfmath is sfadd, which may be more efficient, but is
less versatile.

Examples:

sfmath x=file1.rsf y=file2.rsf power=file3.rsf output='sin((x+2*y)^power)' > out.rsf
sfmath < file1.rsf tau=file2.rsf output='exp(tau*input)' > out.rsf
sfmath n1=100 type=complex output="exp(I*x1)" > out.rsf

Arguments which are not treated as variables in mathematical expressions:
datapath=, type=, out=

See also: sfheadermath.
```

float	**d#=(1,1,...)**	sampling on #-th axis
string	**label=**	data label
string	**label#=**	label on #-th axis
largeint	**n#=**	size of #-th axis
float	**o#=(0,0,...)**	origin on #-th axis
string	**output=**	Mathematical description of the output
string	**type=**	output data type [float,complex]
string	**unit=**	data unit
string	**unit#=**	unit on #-th axis

`sfmath` is a versatile program for mathematical operations with RSF files. It can operate with several input file, all of the same dimensions and data type. The data type can be real (floating point) or complex. Here is an example that demonstrates several features of `sfmath`.

```
bash$ sfmath n1=629 d1=0.01 o1=0 n2=40 d2=1 o2=5 \
output="x2*(8+sin(6*x1+x2/10))" > rad.rsf
bash$ < rad.rsf sfrtoc | sfmath output="input*exp(I*x1)" > rose.rsf
bash$ < rose.rsf sfgraph title=Rose screenratio=1 wantaxis=n | sfpen
```

The first line creates a 2-D dataset that consists of 40 traces 629 samples each. The

values of the data are computed with the formula `"x2*(8+sin(6*x1+x2/10))"`, where `x1` refers to the coordinate on the first axis, and `x2` is the coordinate of the second axis. In the second line, we convert the data from real to complex using `sfrtoc` and produce a complex dataset using formula `"input*exp(I*x1)"`, where `input` refers to the input file. Finally, we plot the complex data as a collection of parametric curves using `sfgraph` and display the result using `sfpen`. The plot appearing on your screen should look similar to Figure 1.

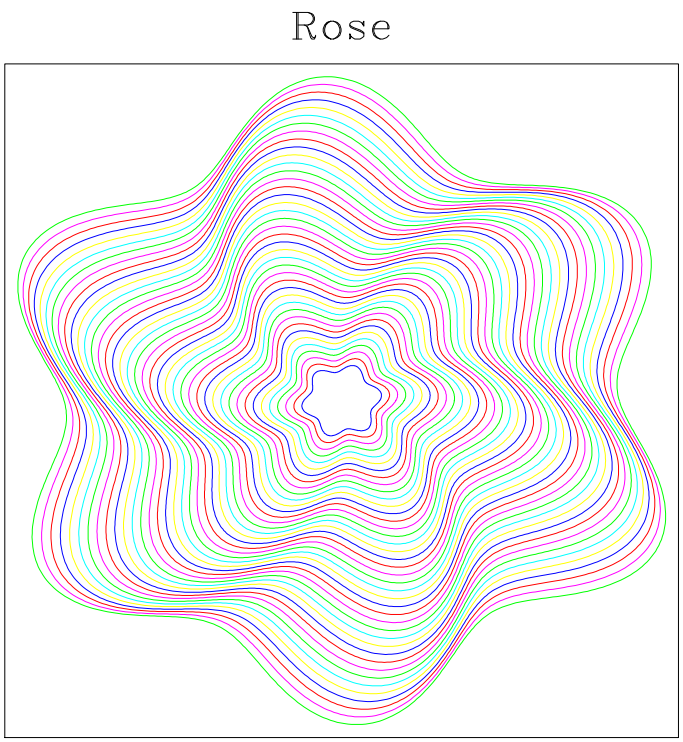

Figure 1: This figure was created with `sfmath`.

One possible alternative to the second line above is

```
bash$ < rad.rsf sfmath output=x1 > ang.rsf
bash$ sfmath r=rad.rsf a=ang.rsf output="r*cos(a)" > cos.rsf
bash$ sfmath r=rad.rsf a=ang.rsf output="r*sin(a)" > sin.rsf
bash$ sfcmplx cos.rsf sin.rsf > rose.rsf
```

Here we refer to input files by names (`r` and `a`) and combine the names in a formula.

sfpad: Pad a dataset with zeros.

```
sfpad < in.rsf > out.rsf beg#=0 end#=0

n\#out is equivalent to n\#, both of them overwrite end\#.
```

int	**beg#=0**	the number of zeros to add before the beginning of #-th axis
int	**end#=0**	the number of zeros to add after the end of #-th axis

pad increases the dimensions of the input dataset by padding the data with zeroes. Here are some simple examples.

```
bash$ sfspike n1=5 n2=3 > one.rsf
bash$ sfdisfil < one.rsf
   0:              1            1            1            1            1
   5:              1            1            1            1            1
  10:              1            1            1            1            1
bash$ < one.rsf sfpad n2=5 | sfdisfil
   0:              1            1            1            1            1
   5:              1            1            1            1            1
  10:              1            1            1            1            1
  15:              0            0            0            0            0
  20:              0            0            0            0            0
bash$ < one.rsf sfpad beg2=2 | sfdisfil
   0:              0            0            0            0            0
   5:              0            0            0            0            0
  10:              1            1            1            1            1
  15:              1            1            1            1            1
  20:              1            1            1            1            1
bash$ < one.rsf sfpad beg2=1 end2=1 | sfdisfil
   0:              0            0            0            0            0
   5:              1            1            1            1            1
  10:              1            1            1            1            1
  15:              1            1            1            1            1
  20:              0            0            0            0            0
bash$ < one.rsf sfwindow n1=3 | sfpad n1=5 n2=5 beg1=1 beg2=1 | sfdisfil
   0:              0            0            0            0            0
   5:              0            1            1            1            0
  10:              0            1            1            1            0
  15:              0            1            1            1            0
  20:              0            0            0            0            0
```

You can use sfcat to pad data with values other than zeroes.

sfput: Input parameters into a header.

```
sfput < in.rsf > out.rsf
```

`sfput` is a very simple program. It simply appends parameters from the command line to the output RSF file. One can achieve similar results with editing by hand or with standard Unix utilities like `sed` and `echo`. `sfput` is sometimes more convenient because it handles input/output operations similarly to other regular RSF programs.

```
bash$ sfspike n1=10 > spike.rsf
bash$ sfin spike.rsf
spike.rsf:
    in="/tmp/spike.rsf@"
    esize=4 type=float form=native
    n1=10          d1=0.004       o1=0          label1="Time" unit1="s"
        10 elements 40 bytes
bash$ sfput < spike.rsf d1=25 label1=Depth unit1=m > spike2.rsf
bash$ sfin spike2.rsf
spike2.rsf:
    in="/tmp/spike2.rsf@"
    esize=4 type=float form=native
    n1=10          d1=25          o1=0          label1="Depth" unit1="m"
        10 elements 40 bytes
```

sfreal: Extract real (sfreal) or imaginary (sfimag) part of a complex dataset.

```
sfreal < cmplx.rsf > real.rsf
```

`sfreal` extracts the real part of a complex type dataset. The imaginary part can be extracted with `sfimag`, an the real and imaginary part can be combined together with `sfcmplx`.

Here is a simple example. Let us first create a complex dataset with `sfmath`

```
bash$ sfmath n1=10 type=complex output="(2+I)*x1" > cmplx.rsf
bash$ fdisfil < cmplx.rsf
   0:          0,         0i         2,         1i         4,         2i
   3:          6,         3i         8,         4i        10,         5i
   6:         12,         6i        14,         7i        16,         8i
   9:         18,         9i
```

Extracting the real part with `sfreal`:

```
bash$ sfreal < cmplx.rsf | sfdisfil
   0:             0             2             4             6             8
   5:            10            12            14            16            18
```

Extracting the imaginary part with `sfimag`:

```
bash$ sfimag < cmplx.rsf | sfdisfil
   0:             0             1             2             3             4
   5:             5             6             7             8             9
```

sfreverse: Reverse one or more axes in the data hypercube.

```
sfreverse < in.rsf > out.rsf which=-1 verb=n memsize=sf_memsize() opt=
```

int	**memsize=sf_memsize()**		Max amount of RAM (in Mb) to be used
string	**opt=**		If y, change o and d parameters on the reversed axis; if i, don't change o and d
bool	**verb=n**	[y/n]	Verbosity flag
int	**which=-1**		Which axis to reverse. To reverse a given axis, start with 0, add 1 to number to reverse n1 dimension, add 2 to number to reverse n2 dimension, add 4 to number to reverse n3 dimension, etc. Thus, which=7 would reverse the first three dimensions, which=5 just n1 and n3, etc. which=0 will just pass the input on through unchanged.

Here is an example of using `sfreverse`. First, let us create a 2-D dataset.

```
bash$ sfmath n1=5 d1=1 n2=3 d2=1 output=x1+x2 > test.rsf
bash$ < test.rsf sfdisfil
   0:             0             1             2             3             4
   5:             1             2             3             4             5
  10:             2             3             4             5             6
```

Reversing the first axis:

```
bash$ < test.rsf sfreverse which=1 | sfdisfil
   0:             4             3             2             1             0
   5:             5             4             3             2             1
  10:             6             5             4             3             2
```

Reversing the second axis:

```
bash$ < test.rsf sfreverse which=2 | sfdisfil
   0:             2             3             4             5             6
   5:             1             2             3             4             5
  10:             0             1             2             3             4
```

Reversing both the first and the second axis:

```
bash$ < test.rsf sfreverse which=3 | sfdisfil
   0:             2             3             4             5             6
   5:             1             2             3             4             5
  10:             0             1             2             3             4
```

As you can see, the `which=` parameter controls the axes that are being reversed by encoding them into one number.

When an axis is reversed, what happens with its axis origin and sampling parameters? This behavior is controlled by `opt=`. In our example,

```
bash$ < test.rsf sfget n1 o1 d1
n1=5
o1=0
d1=1
bash$ < test.rsf sfreverse which=1 | sfget o1 d1
o1=4
d1=-1
```

The default behavior (equivalent to `opt=y`) puts the origin `o1` at the end of the axis and reverses the sampling parameter `d1`. Using `opt=n` preserves the sampling but reverses the origin.

```
bash$ < test.rsf sfreverse which=1 opt=n | sfget o1 d1
o1=-4
d1=1
```

Using `opt=i` preserves both the sampling and the origin while reversing the axis.

```
bash$ < test.rsf sfreverse which=1 opt=i | sfget o1 d1
o1=0
d1=1
```

One of the three possible behaviors may be desirable depending on the application.

sfrm: Remove RSF files together with their data.

```
sfrm file1.rsf [file2.rsf ...]  [-i] [-v] [-f]

Mimics the standard Unix rm command.

See also: sfmv, sfcp.
```

`sfrm` is a program for removing RSF files. Its arguments mimic the arguments of the standard Unix `rm` utility: `-v` for verbosity, `-i` for interactive inquiry, `-f` for force removal of suspicious files. Unlike the Unix `rm`, `sfrm` removes both the RSF header files and the binary files that the headers point to.

Example:

```
bash$ sfspike n1=10 > spike.rsf datapath=./
bash$ sfget in < spike.rsf
in=./spike.rsf@
bash$ ls spike*
spike.rsf  spike.rsf@
bash$ sfrm -v spike.rsf
sfrm: sf_rm: Removing header spike.rsf
sfrm: sf_rm: Removing data ./spike.rsf@
bash$ ls spike*
ls: No match.
```

sfrotate: Rotate a portion of one or more axes in the data hypercube.

```
sfrotate < in.rsf > out.rsf verb=n memsize=sf_memsize() rot#=(0,0,...)
```

int	**memsize=sf_memsize()**		Max amount of RAM (in Mb) to be used
int	**rot#=(0,0,...)**		length of #-th axis that is moved to the end
bool	**verb=n**	[y/n]	Verbosity flag

`sfrotate` modifies the input dataset by splitting it into parts and putting the parts back in a different order. Here is a quick example.

```
bash$ sfmath n1=5 d1=1 n2=3 d2=1 output=x1+x2 > test.rsf
bash$ < test.rsf sfdisfil
   0:            0           1           2           3           4
   5:            1           2           3           4           5
  10:            2           3           4           5           6
```

Rotating the first axis by putting the last two columns in front:

```
bash$ < test.rsf sfrotate rot1=2 | sfdisfil
   0:             3             4             0             1             2
   5:             4             5             1             2             3
  10:             5             6             2             3             4
```

Rotating the second axis by putting the last row in front:

```
bash$ < test.rsf sfrotate rot2=1 | sfdisfil
   0:             2             3             4             5             6
   5:             0             1             2             3             4
  10:             1             2             3             4             5
```

Rotating both the first and the second axis:

```
bash$ < test.rsf sfrotate rot1=3 rot2=1 | sfdisfil
   0:             4             5             6             2             3
   5:             2             3             4             0             1
  10:             3             4             5             1             2
```

The transformation is shown schematically in Figure 2.

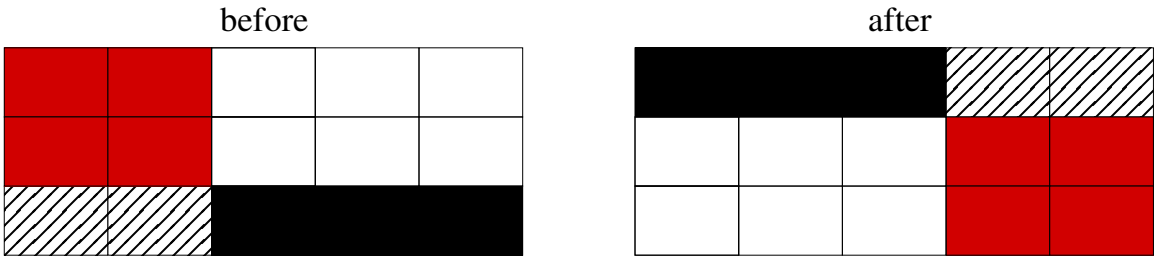

Figure 2: Schematic transformation of data with `sfrotate`.

sfrtoc: Convert real data to complex (by adding zero imaginary part).

```
sfrtoc < real.rsf > cmplx.rsf
```

```
See also: sfcmplx
```

The input to `sfrtoc` can be any `type=float` dataset:

```
bash$ sfspike n1=10 n2=20 n3=30 >real.rsf
bash$ sfin real.rsf
real.rsf:
    in="/var/tmp/real.rsf@"
    esize=4 type=float form=native
    n1=10          d1=0.004       o1=0          label1="Time" unit1="s"
    n2=20          d2=0.1         o2=0          label2="Distance" unit2="km"
    n3=30          d3=0.1         o3=0          label3="Distance" unit3="km"
        6000 elements 24000 bytes
```

The output dataset will have `type=complex`, and its binary will be twice the size of the input:

```
bash$ <real.rsf sfrtoc >complex.rsf
bash$ sfin complex.rsf
complex.rsf:
    in="/var/tmp/complex.rsf@"
    esize=8 type=complex form=native
    n1=10          d1=0.004       o1=0          label1="Time" unit1="s"
    n2=20          d2=0.1         o2=0          label2="Distance" unit2="km"
    n3=30          d3=0.1         o3=0          label3="Distance" unit3="km"
        6000 elements 48000 bytes
```

sfscale: Scale data.

```
sfscale < in.rsf > out.rsf axis=0 rscale=0.  dscale=1.

To scale by a constant factor, you can also use sfmath.
```

int	**axis=0**	Scale by maximum in the dimensions up to this axis.
float	**dscale=1.**	Scale by this factor (works if rscale=0)
float	**rscale=0.**	Scale by this factor.

`sfscale` scales the input dataset by a factor. Here are some simple examples. First, let us create a test dataset.

```
bash$ sfmath n1=5 n2=3 o1=1 o2=1 output="x1*x2" > test.rsf
bash$ < test.rsf sfdisfil
   0:             1            2            3            4            5
   5:             2            4            6            8           10
  10:             3            6            9           12           15
```

Scale every data point by 2:

```
bash$ < test.rsf sfscale dscale=2 | sfdisfil
   0:             2             4             6             8            10
   5:             4             8            12            16            20
  10:             6            12            18            24            30
```

Divide every trace by its maximum value:

```
bash$ < test.rsf sfscale axis=1 | sfdisfil
   0:           0.2           0.4           0.6           0.8             1
   5:           0.2           0.4           0.6           0.8             1
  10:           0.2           0.4           0.6           0.8             1
```

Divide by the maximum value in the whole 2-D dataset:

```
bash$ < test.rsf sfscale axis=2 | sfdisfil
   0:       0.06667        0.1333           0.2        0.2667        0.3333
   5:        0.1333        0.2667           0.4        0.5333        0.6667
  10:           0.2           0.4           0.6           0.8             1
```

The `rscale=` parameter is synonymous to `dscale=` except when it is equal to zero. With `sfscale dscale=0`, the dataset gets multiplied by zero. If using `rscale=0`, the other parameters are used to define scaling. Thus, `sfscale rscale=0 axis=1` is equivalent to `sfscale axis=1`, and `sfscale rscale=0` is equivalent to `sfscale dscale=1`.

sfspike: Generate simple data: spikes, boxes, planes, constants.

```
sfspike < in.rsf > spike.rsf mag= nsp=1 k#=[0,...]  l#=[k1,k2,...]  p#=[0,...]
n#= o#=[0,0,...]  d#=[0.004,0.1,0.1,...]  label#=[Time,Distance,Distance,...]
unit#=[s,km,km,...]  title=

Spike positioning is given in samples and starts with 1.
```

float	**d#=[0.004,0.1,0.1,...]**	sampling on #-th axis
ints	**k#=[0,...]**	spike starting position [nsp]
ints	**l#=[k1,k2,...]**	spike ending position [nsp]
string	**label#=[Time,Distance,Distance,...]**	label on #-th axis
floats	**mag=**	spike magnitudes [nsp]
int	**n#=**	size of #-th axis
int	**nsp=1**	Number of spikes
float	**o#=[0,0,...]**	origin on #-th axis
floats	**p#=[0,...]**	spike inclination (in samples) [nsp]
string	**title=**	title for plots
string	**unit#=[s,km,km,...]**	unit on #-th axis

`sfspike` takes no input and generates an output with "spikes". It is an easy way to create data. Here is an example:

```
bash$ sfspike n1=5 n2=3 k1=4 k2=1 | sfdisfil
   0:             0            0            0            1            0
   5:             0            0            0            0            0
  10:             0            0            0            0            0
```

The spike location is specified by parameters `k1=4` and `k2=1`. Note that the locations are numbered starting from 1. If one of the parameters is omitted or given the value of zero, the spike in the corresponding direction becomes a plane:

```
bash$ sfspike n1=5 n2=3 k1=4 | sfdisfil
   0:             0            0            0            1            0
   5:             0            0            0            1            0
  10:             0            0            0            1            0
```

If no spike parameters are given, the whole dataset is filled with ones:

```
bash$ sfspike n1=5 n2=3 | sfdisfil
   0:             1            1            1            1            1
   5:             1            1            1            1            1
  10:             1            1            1            1            1
```

To create several spikes, use the `nsp=` parameter and give a comma-separated list of values to `k#=` arguments:

```
bash$ sfspike n1=5 n2=3 nsp=3 k1=1,3,4 k2=1,2,3 | sfdisfil
   0:             1            0            0            0            0
   5:             0            0            1            0            0
  10:             0            0            0            1            0
```

If the number of values in the list is smaller than `nsp`, the last value gets repeated, and the spikes add on top of each other, creating larger amplitudes:

```
bash$ sfspike n1=5 n2=3 nsp=3 k1=1,3 k2=1,2 | sfdisfil
   0:             1            0            0            0            0
   5:             0            0            2            0            0
  10:             0            0            0            0            0
```

The magnitude of the spikes can be controlled explicitly with the `mag=` parameter:

```
bash$ sfspike n1=5 n2=3 nsp=3 k1=1,3,4 k2=1,2,3 mag=1,4,2 | sfdisfil
   0:             1            0            0            0            0
   5:             0            0            4            0            0
  10:             0            0            0            2            0
```

You can create boxes instead of spikes by using `l#=` parameters:

```
bash$ sfspike n1=5 n2=3 k1=2 l1=4 k2=2 mag=8 | sfdisfil
   0:             0            0            0            0            0
   5:             0            8            8            8            0
  10:             0            0            0            0            0
```

In this case, `k1=2` specifies the box start, and `l1=4` specifies the box end.

Finally, multi-dimensional planes can be given an inclination by using `p#=` parameters:

```
bash$ sfspike n1=5 n2=3 k1=2 p2=1 | sfdisfil
   0:             0            1            0            0            0
   5:             0            0            1            0            0
  10:             0            0            0            1            0
```

When the inclination value is not integer, simple linear interpolation is used:

```
bash$ sfspike n1=5 n2=3 k1=2 p2=0.7 | sfdisfil
   0:             0            1            0            0            0
   5:             0          0.3          0.7            0            0
  10:             0            0          0.6          0.4            0
```

`sfspike` supplies default dimensions and labels to all axis:

```
bash$ sfspike n1=5 n2=3 n3=4 > spike.rsf
bash$ sfin spike.rsf
spike.rsf:
    in="/var/tmp/spike.rsf@"
    esize=4 type=float form=native
    n1=5           d1=0.004       o1=0          label1="Time" unit1="s"
    n2=3           d2=0.1         o2=0          label2="Distance" unit2="km"
    n3=4           d3=0.1         o3=0          label3="Distance" unit3="km"
60 elements 240 bytes
```

As you can see, the first axis is assumed to be time, with sampling of 0.004 seconds. All other axes are assumed to be distance, with sampling of 0.1 kilometers. All these parameters can be changed on the command line.

```
bash$ sfspike n1=5 n2=3 n3=4 label3=Offset unit3=ft d3=20 > spike.rsf
bash$ sfin spike.rsf
spike.rsf:
    in="/var/tmp/spike.rsf@"
    esize=4 type=float form=native
    n1=5            d1=0.004        o1=0            label1="Time" unit1="s"
    n2=3            d2=0.1          o2=0            label2="Distance" unit2="km"
    n3=4            d3=20           o3=0            label3="Offset" unit3="ft"
60 elements 240 bytes
```

sfspray: Extend a dataset by duplicating in the specified axis dimension.

```
sfspray < in.rsf > out.rsf axis=2 n= d= o= label= unit=
```

This operation is adjoint to sfstack.

int	**axis=2**	which axis to spray
float	**d=**	Sampling of the newly created dimension
string	**label=**	Label of the newly created dimension
int	**n=**	Size of the newly created dimension
float	**o=**	Origin of the newly created dimension
string	**unit=**	Units of the newly created dimension

`sfspray` extends the input hypercube by replicating the data in one of the dimensions. The output dataset acquires one additional dimension. Here is an example:

Start with a 2-D dataset

```
bash$ sfmath n1=5 n2=2 output=x1+x2 > test.rsf
bash$ sfin test.rsf
test.rsf:
    in="/var/tmp/test.rsf@"
    esize=4 type=float form=native
    n1=5            d1=1            o1=0
    n2=2            d2=1            o2=0
        10 elements 40 bytes
bash$ < test.rsf sfdisfil
   0:             0            1            2            3            4
   5:             1            2            3            4            5
```

Extend the data in the second dimension

```
bash$ < test.rsf sfspray axis=2 n=3 > test2.rsf
bash$ sfin test2.rsf
```

```
test2.rsf:
    in="/var/tmp/test2.rsf@"
    esize=4 type=float form=native
    n1=5            d1=1            o1=0
    n2=3            d2=1            o2=0
    n3=2            d3=1            o3=0
        30 elements 120 bytes
bash$ < test2.rsf sfdisfil
   0:             0            1            2            3            4
   5:             0            1            2            3            4
  10:             0            1            2            3            4
  15:             1            2            3            4            5
  20:             1            2            3            4            5
  25:             1            2            3            4            5
```

The output is three-dimensional, with traces from the original data duplicated along the second axis.

Extend the data in the third dimension

```
bash$ < test.rsf sfspray axis=3 n=2 > test3.rsf
bash$ sfin test3.rsf
test3.rsf:
    in="/var/tmp/test3.rsf@"
    esize=4 type=float form=native
    n1=5            d1=1            o1=0
    n2=2            d2=1            o2=0
    n3=2            d3=?            o3=?
        20 elements 80 bytes
bash$ < test3.rsf sfdisfil
   0:             0            1            2            3            4
   5:             1            2            3            4            5
  10:             0            1            2            3            4
  15:             1            2            3            4            5
```

The output is also three-dimensional, with the original data replicated along the third axis.

sfstack: Stack a dataset over one of the dimensions.

```
sfstack < in.rsf > out.rsf scale= axis=2 rms=n norm=y min=n max=n prod=n

This operation is adjoint to sfspray.
```

int	**axis=2**		which axis to stack
bool	**max=n**	[y/n]	If y, find maximum instead of stack. Ignores rms and norm.
bool	**min=n**	[y/n]	If y, find minimum instead of stack. Ignores rms and norm.
bool	**norm=y**	[y/n]	If y, normalize by fold.
bool	**prod=n**	[y/n]	If y, find product instead of stack. Ignores rms and norm.
bool	**rms=n**	[y/n]	If y, compute the root-mean-square instead of stack.
floats	**scale=**		optionally scale before stacking [n2]

While `sfspray` adds a dimension to a hypercube, `sfstack` effectively removes one of the dimensions by stacking over it. Here are some examples:

```
bash$ sfmath n1=5 n2=3 output=x1+x2 > test.rsf
bash$ < test.rsf sfdisfil
   0:             0            1            2            3            4
   5:             1            2            3            4            5
  10:             2            3            4            5            6
bash$ < test.rsf sfstack axis=2 | sfdisfil
   0:           1.5            2            3            4            5
bash$ < test.rsf sfstack axis=1 | sfdisfil
   0:           2.5            3            4
```

Why is the first value not 1 (in the first case) or 2 (in the second case)? By default, `sfstack` normalizes the stack by the fold (the number of non-zero entries). To avoid normalization, use `norm=n`, as follows:

```
bash$ < test.rsf sfstack norm=n | sfdisfil
   0:             3            6            9           12           15
```

`sfstack` can also compute root-mean-square values as well as minimum and maximum values.

```
bash$ < test.rsf sfstack rms=y | sfdisfil
   0:         1.581         2.16        3.109        4.082        5.066
bash$ < test.rsf sfstack min=y | sfdisfil
   0:             0            1            2            3            4
bash$ < test.rsf sfstack axis=1 max=y | sfdisfil
   0:             4            5            6
```

sftransp: Transpose two axes in a dataset.

```
sftransp < in.rsf > out.rsf memsize=sf_memsize() plane=

If you get a "Cannot allocate memory" error, give the program a
memsize=1 command-line parameter to force out-of-core operation.
```

int	**memsize=sf_memsize()**	Max amount of RAM (in Mb) to be used
int	**plane=**	Two-digit number with axes to transpose. The default is 12

The `sftransp` program transposes the input hypercube exchanging the two axes specified by the `plane=` parameter.

```
bash$ sfspike n1=10 n2=20 n3=30 > orig123.rsf
bash$ sfin orig123.rsf
orig123.rsf:
    in="/var/tmp/orig123.rsf@"
    esize=4 type=float form=native
    n1=10           d1=0.004        o1=0           label1="Time" unit1="s"
    n2=20           d2=0.1          o2=0           label2="Distance2" unit2="km"
    n3=30           d3=0.1          o3=0           label3="Distance3" unit3="km"
        6000 elements 24000 bytes
bash$ <orig123.rsf sftransp plane=23 >out132.rsf
bash$ sfin out132.rsf
out132.rsf:
    in="/var/tmp/out132.rsf@"
    esize=4 type=float form=native
    n1=10           d1=0.004        o1=0           label1="Time" unit1="s"
    n2=30           d2=0.1          o2=0           label2="Distance3" unit2="km"
    n3=20           d3=0.1          o3=0           label3="Distance2" unit3="km"
        6000 elements 24000 bytes
bash$ <orig123.rsf sftransp plane=13 >out321.rsf
bash$ sfin out321.rsf
out321.rsf:
    in="/var/tmp/out132.rsf@"
    esize=4 type=float form=native
    n1=30           d1=0.1          o1=0           label1="Distance" unit1="km"
    n2=20           d2=0.1          o2=0           label2="Distance" unit2="km"
    n3=10           d3=0.004        o3=0           label3="Time" unit3="s"
        6000 elements 24000 bytes
```

`sftransp` tries to fit the dataset in memory to transpose it there but, if not enough memory is available, it performs a slower transpose out of core using disk operations. You can control the amount of available memory using the `memsize=` parameter or

the RSFMEMSIZE environmental variable.

sfwindow: Window a portion of a dataset.

```
sfwindow < in.rsf > out.rsf verb=n squeeze=y j#=(1,...)  d#=(d1,d2,...)
f#=(0,...)  min#=(o1,o2,,...)  n#=(0,...)  max#=(o1+(n1-1)*d1,o2+(n1-1)*d2,,...)
```

float	**d#=(d1,d2,...)**		sampling in #-th dimension
largeint	**f#=(0,...)**		window start in #-th dimension
int	**j#=(1,...)**		jump in #-th dimension
float	**max#=(o1+(n1-1)*d1,o2+(n1-1)*d2,,...)**		maximum in #-th dimension
float	**min#=(o1,o2,,...)**		minimum in #-th dimension
largeint	**n#=(0,...)**		window size in #-th dimension
bool	**squeeze=y**	[y/n]	if y, squeeze dimensions equal to 1 to the end
bool	**verb=n**	[y/n]	Verbosity flag

sfwindow is used to window a portion of the dataset. Here is a quick example: Start by creating some data.

```
bash$ sfmath n1=5 n2=3 o1=1 o2=1 output="x1*x2" > test.rsf
bash$ < test.rsf sfdisfil
   0:             1            2            3            4            5
   5:             2            4            6            8           10
  10:             3            6            9           12           15
```

Now window the first two rows:

```
bash$ < test.rsf sfwindow n2=2 | sfdisfil
   0:             1            2            3            4            5
   5:             2            4            6            8           10
```

Window the first three columns:

```
bash$ < test.rsf sfwindow n1=3 | sfdisfil
   0:             1            2            3            2            4
   5:             6            3            6            9
```

Window the middle row:

```
bash$ < test.rsf sfwindow f2=1 n2=1 | sfdisfil
   0:             2            4            6            8           10
```

You can interpret the f# and n# parameters as meaning "skip that many rows/-columns" and "select that many rows/columns" correspondingly. Window the middle point in the dataset:

```
bash$ < test.rsf sfwindow f1=2 n1=1 f2=1 n2=1 | sfdisfil
   0:             6
```

Window every other column:

```
bash$ < test.rsf sfwindow j1=2 | sfdisfil
   0:             1            3            5            2            6
   5:            10            3            9           15
```

Window every third column:

```
bash$ < test.rsf sfwindow j1=3 | sfdisfil
   0:             1            4            2            8            3
   5:            12
```

Alternatively, `sfwindow` can use the minimum and maximum parameters to select a window. In the following example, we are creating a dataset with `sfspike` and then windowing a portion of it between 1 and 2 seconds in time and sampled at 8 miliseconds.

```
bash$ sfspike n1=1000 n2=10 > spike.rsf
bash$ sfin spike.rsf
spike.rsf:
    in="/var/tmp/spike.rsf@"
    esize=4 type=float form=native
    n1=1000        d1=0.004       o1=0          label1="Time" unit1="s"
    n2=10          d2=0.1         o2=0          label2="Distance" unit2="km"
        10000 elements 40000 bytes
bash$ < spike.rsf sfwindow min1=1 max1=2 d1=0.008 > window.rsf
bash$ sfin window.rsfwindow.rsf:
    in="/var/tmp/window.rsf@"
    esize=4 type=float form=native
    n1=126         d1=0.008       o1=1          label1="Time" unit1="s"
    n2=10          d2=0.1         o2=0          label2="Distance" unit2="km"
        1260 elements 5040 bytes
```

By default, `sfwindow` "squeezes" the hypercube dimensions that are equal to one toward the end of the dataset. Here is an example of taking a time slice:

```
bash$ < spike.rsf sfwindow n1=1 min1=1 > slice.rsf
bash$ sfin slice.rsf
slice.rsf:
    in="/var/tmp/slice.rsf@"
```

```
    esize=4 type=float form=native
    n1=10           d1=0.1          o1=0            label1="Distance" unit1="km"
    n2=1            d2=0.004        o2=1            label2="Time" unit2="s"
        10 elements 40 bytes
```

You can change this behavior by specifying `squeeze=n`.

```
bash$ < spike.rsf sfwindow n1=1 min1=1 squeeze=n > slice.rsf
bash$ sfin slice.rsf slice.rsf:
    in="/var/tmp/slice.rsf@"
    esize=4 type=float form=native
    n1=1            d1=0.004        o1=1            label1="Time" unit1="s"
    n2=10           d2=0.1          o2=0            label2="Distance" unit2="km"
        10 elements 40 bytes
```

REFERENCES

Claerbout, J., 1998, Multidimensional recursive filters via a helix: Geophysics, **63**, 1532–1541.