Madagascar Programming Reference Manual

Seismic Wave Analysis Group
`http://swag.kaust.edu.sa`

KAUST

Mohammad Akbar Zuberi
Tariq Alkhalifah

Contents

Preface

This document is a contribution to the Madagascar developers community and it is meant to help out with Programming in Madagascar. It also came out of our selfish need to learn more about the libraries described in Madagascar. This document specifically lists and describes the RSF API functions supported by Madagascar for C.

The first chapter is an introduction, summarizing the key information needed to understand Madagascar and its history. In the second chapter there is an example program with explanation for every line of the code. The example program is a finite difference modeling code.

The RSF Function Library starts from Chapter 3, which is a description of the data types used in Madagascar. Some data types are defined in Madagascar and some are from the standard C library headers.

The subsequent chapters group different subroutines (the .c files) which are used for a particular task. Chapter 4 lists the .c files which have the functions used in the preparation for input, such as `sf_alloc`, which allocates the required space.

Chapter 5 is related to the handling of the `.rsf` files, for example file input/output, extracting or inserting a parameter from a file.

Chapter 6 is for the error handling subroutines. It lists the functions which print the required error messages.

Chapter 7 lists the linear operators. There is a detailed introduction in the beginning of the chapter.

Chapter 8 is for data analysis subroutines, for example `kiss_fftr.c` for the Fourier inverse and forward transform of real time signals.

Chapter 9 lists the subroutines for the filtering and convolution.

Chapter 10 lists the solvers. It has the subroutines for tasks like solving a first order ODE using the Runge-Kutta solver. There are subroutines in this chapter which perform the iterations for the conjugate gradient method for real and complex data. Some other functions include the root finder and tridagonal matrix solver.

Chapter 11 is for the interpolation subroutines. It has subroutines for 1D, 2D and 3D interpolation. There are functions for B-Spline interpolation, calculation for B-spline coefficients, ENO and ENO power-p interpolation.

Chapter 12 has the subroutines for of smoothing and edge detection. It has functions for 1D and 2D triangle smoothing.

Chapter 13 lists the Ray Tracing functions.

Chapter 14 has some general purpose tools like functions for evaluating a mathematical expression and generating random numbers.

Chapter 15 is concerned with the geometry. It has the functions which define vectors and points which can be used to define the source and receiver coordinates. It also has the axa.c file which defines the functions for creating and operating on axes.

Chapter 16 is for the miscellaneous functions.

Chapter 17 lists the function which are system specific for example system.c file defines functions which can run a given command on the terminal from within the program.

Chapter 1

Introduction

Madagascar provides interfaces for programming in C and optionally in C++ and Fortran90, etc. The C API is installed automatically during the installation of the Madagascar package. The other interfaces should be installed separately, if necessary, as explained in the installation guide.

Programming using a language for which an API (Applications Programming Interface) is provided by Madagascar allows the user to operate on RSF files and also use some predefined functions from the RSF library. These library files are located in the directories `api` and `build/api` in the source directory.

A guide to Madagascar programming interface can be found at the Madagascar website

Chapter 2

An example: Finite-Difference modeling

To demonstrate the use of the RSF library, a time-domain finite-difference modeling program is explained in detail.

2.1 Introduction

This section presents time-domain finite-difference modeling [1] written with the RSF library. The program is demonstrated with the C, C++ and Fortran 90 interfaces. The acoustic wave-equation

$$\Delta U - \frac{1}{v^2}\frac{\partial^2 U}{\partial t^2} = f(t)$$

can be written as

$$|\Delta U - f(t)|v^2 = \frac{\partial^2 U}{\partial t^2},$$

where Δ is the Laplacian symbol, $f(t)$ is the source wavelet, v is the velocity, and U is a scalar wavefield. A discrete time-step involves the following computations:

$$U_{i+1} = [\Delta U - f(t)]v^2 \Delta t^2 + 2U_i - U_{i-1},$$

where $U_{i?1}$, U_i and U_{i+1} represent the propagating wavefield at various time steps. In this exercise we shall use a discrete Laplacian accurate up to the fourth order and the second derivative of time is accurate up to the second order.

2.2 C program

```
/* time-domain acoustic FD modeling */
#include <rsf.h>

int main(int argc, char* argv[])
```

```
{
    /* Laplacian coefficients */
    float c0=-30./12.,c1=+16./12.,c2=- 1./12.;

    bool verb;           /* verbose flag */
    sf_file Fw=NULL,Fv=NULL,Fr=NULL,Fo=NULL; /* I/O files */
    sf_axis at,az,ax;    /* cube axes */
    int it,iz,ix;        /* index variables */
    int nt,nz,nx;
    float dt,dz,dx,idx,idz,dt2;

    float  *ww,**vv,**rr;     /* I/O arrays*/
    float **um,**uo,**up,**ud;/* tmp arrays */

    sf_init(argc,argv);
    if(! sf_getbool("verb",&verb)) verb=0;

    /* setup I/O files */
    Fw = sf_input ("in" );
    Fo = sf_output("out");
    Fv = sf_input ("vel");
    Fr = sf_input ("ref");

    /* Read/Write axes */
    at = sf_iaxa(Fw,1); nt = sf_n(at); dt = sf_d(at);
    az = sf_iaxa(Fv,1); nz = sf_n(az); dz = sf_d(az);
    ax = sf_iaxa(Fv,2); nx = sf_n(ax); dx = sf_d(ax);

    sf_oaxa(Fo,az,1);
    sf_oaxa(Fo,ax,2);
    sf_oaxa(Fo,at,3);

    dt2 =    dt*dt;
    idz = 1/(dz*dz);
    idx = 1/(dx*dx);

    /* read wavelet, velocity & reflectivity */
    ww = sf_floatalloc(nt);     sf_floatread(ww   ,nt   ,Fw);
    vv = sf_floatalloc2(nz,nx); sf_floatread(vv[0],nz*nx,Fv);
    rr = sf_floatalloc2(nz,nx); sf_floatread(rr[0],nz*nx,Fr);

    /* allocate temporary arrays */
    um = sf_floatalloc2(nz,nx);
    uo = sf_floatalloc2(nz,nx);
    up = sf_floatalloc2(nz,nx);
    ud = sf_floatalloc2(nz,nx);

    for (iz=0; iz<nz; iz++) {
        for (ix=0; ix<nx; ix++) {
            um[ix][iz]=0;
            uo[ix][iz]=0;
            up[ix][iz]=0;
            ud[ix][iz]=0;
        }
    }

    /* MAIN LOOP */
    if(verb) fprintf(stderr,"\n");
    for (it=0; it<nt; it++) {
        if(verb) fprintf(stderr,"\b\b\b\b\b%d",it);

         /* 4th order laplacian */
         for (iz=2; iz<nz-2; iz++) {
             for (ix=2; ix<nx-2; ix++) {
                 ud[ix][iz] =
                   c0* uo[ix  ][iz  ] * (idx+idz) +
                   c1*(uo[ix-1][iz  ] + uo[ix+1][iz  ])*idx +
                   c2*(uo[ix-2][iz  ] + uo[ix+2][iz  ])*idx +
                   c1*(uo[ix  ][iz-1] + uo[ix  ][iz+1])*idz +
                   c2*(uo[ix  ][iz-2] + uo[ix  ][iz+2])*idz;
             }
         }

         /* inject wavelet */
         for (iz=0; iz<nz; iz++) {
             for (ix=0; ix<nx; ix++) {
                 ud[ix][iz] -= ww[it] * rr[ix][iz];
             }
         }

         /* scale by velocity */
         for (iz=0; iz<nz; iz++) {
             for (ix=0; ix<nx; ix++) {
                 ud[ix][iz] *= vv[ix][iz]*vv[ix][iz];
             }
         }

         /* time step */
         for (iz=0; iz<nz; iz++) {
             for (ix=0; ix<nx; ix++) {
                 up[ix][iz] = 2*uo[ix][iz]
                              - um[ix][iz]
                              + ud[ix][iz] * dt2;

                 um[ix][iz] = uo[ix][iz];
                 uo[ix][iz] = up[ix][iz];
               }
           }
```

```
103
104             /* write wavefield to output */
105             sf_floatwrite(uo[0],nz*nx,Fo);
106         }
107         if(verb) fprintf(stderr,"\n");
108         sf_close()
109         exit(0);
110     }
```

2.3 Explanation of the code

2-4:

```
2    #include <rsf.h>
3
4    int main(int argc, char* argv[])
```

Line 2 is a preprocessor directive to include the `rsf.h` header file which contains the RSF library functions.

Line 4 has parameters in the main function. This is to enable the program to take command line arguments. `char* argv[]` defines the pointer to the array of type `char` and `int argc` is the length of that array.

7:

```
7        float c0=-30./12.,c1=+16./12.,c2=- 1./12.;
```

As was mentioned earlier, the Laplacian is being evaluated with an accuracy of up to the fourth order. These coefficients arise as a result of using five terms in the discrete form of the Laplacian.

9-14:

```
 9       bool verb;           /* verbose flag */
10       sf_file Fw=NULL,Fv=NULL,Fr=NULL,Fo=NULL; /* I/O files */
11       sf_axis at,az,ax;    /* cube axes */
12       int it,iz,ix;        /* index variables */
13       int nt,nz,nx;
14       float dt,dz,dx,idx,idz,dt2;
```

Line 9 defines a variable `verb` of type `bool`. This variable will be used in the program to check for verbosity flag. Lines 10-11 define the variables of the abstract data type provided by the RSF API. These will be used to store the input and output files. Lines 12-14 are the variables of integer and float type defined to be used as running variables (`it`, `iz`, `ix`) for the main loop, length of the axes (`nt`, `nz`, `nx`), the sampling of the axes (`dt`, `dx`, `dz`) and the squares and inverse squares of the samples (`dt2`, `idz`, `idx`).

16-17:

```
16       float  *ww,**vv,**rr;     /* I/O arrays*/
17       float **um,**uo,**up,**ud;/* tmp arrays */
```

Lines 16-17 define pointers to the arrays, which will be used for `input (*ww , **vv , **rr)` and for temporary storage `(**um,**uo,**up,**ud)`.

19-20:

```
19        sf_init(argc,argv);
20        if(! sf_getbool("verb",&verb)) verb=0;
```

Line 19 initializes the symbol tables used to store the argument from the command line.

Line 20 tests the verbosity flag specified in the command line arguments. If the verbosity flag in the command line is set to n, the variable verb (of type bool) is set to zero. This would allow the verbose output to be printed only if the user set the verbosity flag to y in the command line.

22-26:
```
22        /* setup I/O files */
23       Fw = sf_input ("in" );
24       Fo = sf_output("out");
25       Fv = sf_input ("vel");
26       Fr = sf_input ("ref");
```

In these lines we use the sf_input (see p. 62) and sf_output (see p. 62) functions of the RSF API. These functions take a string as argument and return a variable of type sf_file, we had already defined this type of variables earlier in the program.

28-32:
```
28         /* Read/Write axes */
29         at = sf_iaxa(Fw,1); nt = sf_n(at); dt = sf_d(at);
30         az = sf_iaxa(Fv,1); nz = sf_n(az); dz = sf_d(az);
31         ax = sf_iaxa(Fv,2); nx = sf_n(ax); dx = sf_d(ax);
```

Here we input axes (at, az, ax) using sf_iaxa (p. 261 of the RSF API. sf_iaxa accepts a variables of type sf_file (RSF API) and an integer. The first argument in sf_iaxa is the input file and the second is the axis which we want to input. In the second column we use sf_n (p. 262)from RSF API to get the lengths of the respective axes.

In the third column we use sf_d (p. 263) of the RSF API to get the sampling interval of the respective axes.

33-35:
```
33         sf_oaxa(Fo,az,1);
34         sf_oaxa(Fo,ax,2);
35         sf_oaxa(Fo,at,3);
```

Here we output axes (at, az, ax) using sf_oaxa (p. 262) of the RSF API. sf_oaxa accepts variables of type sf_file (RSF API), sf_axis (RSF API) and an integer. First argument is the output file, second argument is the name of the axis which we want to output and the third is the number of the axis in the output file (n1 is the fastest axis).

37-39:
```
37       dt2 =    dt*dt;
38       idz = 1/(dz*dz);
39       idx = 1/(dx*dx);
```

These lines define the square of the time sampling interval(dt2) and the inverse squares of the sampling interval of the spatial axes.

41-44:
```
41       /* read wavelet, velocity & reflectivity */
42       ww = sf_floatalloc(nt);     sf_floatread(ww   ,nt   ,Fw);
43       vv = sf_floatalloc2(nz,nx); sf_floatread(vv[0],nz*nx,Fv);
44       rr = sf_floatalloc2(nz,nx); sf_floatread(rr[0],nz*nx,Fr);
```

In the first column we allocate the memory required to hold the input wavelet, velocity and reflectivity. This is done using sf_floatalloc (p. 31) and sf_floatalloc2 (p. 35) of the RSF API. sf_floatalloc takes integers as arguments and from these integers it calculates an allocates a block of memory of appropriate size. sf_floatalloc2 is the same as sf_floatalloc except for the fact that the former allocates an array of two dimensions, size of the memory block assigned in this case is the product of the two integers given as arguments (e.g. `nz*nx` in this case).

Then `sf_floatread` (p. 83) of the RSF API is used to read the data from the files into the allocated memory blocks (arrays). The `sf_floatread` takes the arrays, integers and files as arguments and returns arrays filled with the data from the files.

46-50:
```
46        /* allocate temporary arrays */
47        um = sf_floatalloc2(nz,nx);
48        uo = sf_floatalloc2(nz,nx);
49        up = sf_floatalloc2(nz,nx);
50        ud = sf_floatalloc2(nz,nx);
```

Just like the memory blocks were allocated for input files to be read in to, we now allocate memory for the temporary arrays which will be used just for the calculation, using `sf_floatalloc2` (p. 35).

52-59:
```
52        for (iz=0; iz<nz; iz++) {
53            for (ix=0; ix<nx; ix++) {
54                um[ix][iz]=0;
55                uo[ix][iz]=0;
56                up[ix][iz]=0;
57                ud[ix][iz]=0;
58            }
59        }
```

Lines 52-59 initialize the temporary arrays by assigning `0` to each element of every array.

61-64:
```
61        /* MAIN LOOP */
62        if(verb) fprintf(stderr,"\n");
63        for (it=0; it<nt; it++) {
64            if(verb) fprintf(stderr,"\b\b\b\b\b%d",it);
```

Now the main loop starts. The if condition in line 61 prints the message specified in the `fprintf` argument. The `stderr` is a stream in C which is used to direct the output to the screen. In this case the input is just an escape sequence
`n`, which will bring the cursor to the next line if the user opted `y` or `1` to verbose flag in the command line (`verb=y` of `verb=1`).

Then the loop over time starts. Right after the for statement (within the body of the loop) there is another if condition like the first one but this time it prints the the current value of `it`. This has escape sequence_occurring several times. This is when the loop starts the value of `it` which is `0`, is printed on the screen, when the loop returns to the start the new value of it is `1`, so
`b` (backspace) removes the previous value `0`, which is already on the screen, and puts `1` instead.

66-76:
```
66            /* 4th order laplacian */
67            for (iz=2; iz<nz-2; iz++) {
68                for (ix=2; ix<nx-2; ix++) {
69                    ud[ix][iz] =
70                      c0* uo[ix  ][iz  ] * (idx+idz) +
71                      c1*(uo[ix-1][iz  ] + uo[ix+1][iz  ])*idx +
72                      c2*(uo[ix-2][iz  ] + uo[ix+2][iz  ])*idx +
73                      c1*(uo[ix  ][iz-1] + uo[ix  ][iz+1])*idz +
74                      c2*(uo[ix  ][iz-2] + uo[ix  ][iz+2])*idz;
75                 }
76             }
```

This is the calculation for the fourth order laplacian. By the term "4th order" we mean the order of the approximation not the order of the PDE itself which of course is a second order PDE. A second order partial derivative discretized to second order approximation is written as:

$$\frac{\partial^2 U}{\partial x^2} = \frac{U_{i+1} + 2U_i + U_{i-1}}{\Delta x^2}$$

This is the central difference formula for the second order partial derivative with pivot at $i-$th value of U.Similarly for the z direction we have:

$$\frac{\partial^2 U}{\partial z^2} = \frac{U_{i+1} + 2U_i + U_{i-1}}{\Delta z^2}$$

By adding these two we get the central difference formula accurate to the second order for the Laplacian. But we are using a central difference accurate up to the fourth order so for that we have:

$$\frac{\partial^2 U}{\partial x^2} = \frac{1}{\Delta x^2}\left[-\frac{1}{12}U_{i+2} + \frac{16}{12}U_{i+1} - \frac{30}{12}U_i + \frac{16}{12}U_{i-i} - \frac{1}{12}U_{i-2}\right]$$

By writing down a similar equation for z and adding the two we get the fourth order approximation of the Laplacian or as we refer to it here "4th order laplacian".

Now returning back to the code, the first line is the start of the loop in the z direction. Within the body of the `z` loop there is another loop which runs through all the values of `x` for one value of `z`. The second line is start of the for-loop for the `x` direction.

Then in the body of the loop for x direction we use the 2×2 arrays which we defined earlier. This is just the equation of the Laplacian accurate up to the fourth order, as discussed above, with the common coefficients factored out. Note that the loops for `x` and `z` start two units after 0 and end two units before `nx` and `nz`. This is because to evaluate the Laplacian at a particular point (x, z) the farthest values which we are using are two units behind and two units ahead of the current point (x, z) if we include the points `iz=0,1 ; iz=nz-1, nz` and `ix=0,1;ix=nx-1,nx` we will run out of bounds. To fill these we will need a boundary condition which we will get from the next loop for inserting the wavelet.

78-83:
```
78             /* inject wavelet */
79             for (iz=0; iz<nz; iz++) {
80                 for (ix=0; ix<nx; ix++) {
81                     ud[ix][iz] -= ww[it] * rr[ix][iz];
82                 }
83             }
```

These lines insert the wavelet, which means evaluating the expression $\Delta U - f(t)$. ΔU was already calculated in the previous loop and is stored as the array `ud`. `ww` is the array of the wavelet but before subtracting it form the Laplacian (`ud`) we multiply the wavelet amplitude at current time with the reflectivity at every point in space (x, z). This amounts to an initial condition:

$$f(x, z, 0) = g(x, z) = ww(0)rr(x, z),$$

and thus serves the purpose of filling the values at `ix,iz=0` and `ix-2,ix-1=0` and `iz-2,iz-1=0`.

But the source wavelet is not an ideal impulse so it has amplitudes at future times so for each time the wavelet will be multiplied by the reflectivity at every point (x, z). Why multiply the wavelet with reflectivity? Well, this model assumes a hypothetical situation that the source was set off at each and every point in space (x, z) under consideration and scaled by the reflectivity at that point (x, z). What this means is that the source was set off at all the points where there is a change in the acoustic impedance (because reflectivity is the ratio of the difference and sum of the acoustic impedances across an interface).

85-90:

```
 85            /* scale by velocity */
 86            for (iz=0; iz<nz; iz++) {
 87                for (ix=0; ix<nx; ix++) {
 88                    ud[ix][iz] *= vv[ix][iz]*vv[ix][iz];
 89                }
 90            }
```

Here we just multiply $\Delta U - f(t)$ by the velocity, that is, we evaluate $(\Delta U - f(t))v^2$

92-102

```
 92            /* time step */
 93            for (iz=0; iz<nz; iz++) {
 94                for (ix=0; ix<nx; ix++) {
 95                    up[ix][iz] = 2*uo[ix][iz]
 96                               - um[ix][iz]
 97                               + ud[ix][iz] * dt2;
 98
 99                    um[ix][iz] = uo[ix][iz];
100                    uo[ix][iz] = up[ix][iz];
101                  }
102              }
```

Here we calculate the time step, that is,

$$U_{i+1} = [\Delta U - f(t)]v^2\Delta t^2 + 2U_i - U_{i-1}.$$

The first for-loop is for the z direction and within the body of this loop is another for-loop for the x direction. `up` is the array which holds the amplitude of the wave at the current time in the time loop. `uo` is the array which contains the amplitude at a time one unit before the current time and the array um holds the amplitude two units before. `ud` is the array we calculated earlier in the program, now it gets multiplied by Δt^2 (`dt2`) and included in the final equation. This completes the calculation for one value of `it`. Now the arrays need to be updated to represent the next time step. This is done in the last two: The first one says $U_{i-1} \to U_i$ and the second one says $U_i \to U_{i+1}$, that is, the array um is updated by `uo` and then the array `uo` itself gets updated by `up`.

104-106:

```
104              /* write wavefield to output */
105              sf_floatwrite(uo[0],nz*nx,Fo);
106         }
```

After the calculations for one time step are complete we write the array `uo` (remember that `uo` was made equal to `up`, which is the current time step, in the previous line). To write the array in the output file we use `sf_floatwrite` (p. 83) exactly the same way we used `sf_floatread` to read in from the input files, only difference is that the array given as the argument is written into the file given in the last argument. The bracket close is for the time loop, after this the time loop will start all over again for the next time value.

107-109:

```
107         if(verb) fprintf(stderr,"\n");
108         sf_close()
109         exit(0);
110     }
```

The first line puts the cursor in the new line on the screen after the time loop has run through all the time values.

The second line uses `sf_close` (p. 86) from RSF API to remove the temporary files.

The third line uses the `exit()` function in C language to close the streams and return the control to the host environment. The `0` in the argument indicates normal termination of the program. The last bracket closes the main function.

Chapter 3

Data types

3.1 Data types

This chapter contains the descriptions of the data types used in the RSF API.

3.1.1 Complex numbers and FFT

This section lists the data types for the complex numbers and FFT.

kiss_fft_scalar

This is a data type, which defines a scalar real value for the data type `kiss_fft_cpx` for complex numbers. It can be either of type `short` or `float`. Default is `float`.

kiss_fft_cpx

This is a data type (a C structure), which defines a complex number. It has the real and imaginary parts of the complex numbers defined to be of type `kiss_fft_scalar`.

kiss_fft_cfg

This is an object of type `kiss_fft_state` (which is a C data structure).

kiss_fft_state

The `kiss_fft_state` is a data type which defines the required variables for the Fourier transform and allocates the required space. For example the variable inverse of type int indicates whether the transform needs to be an inverse or forward.

kiss_fftr_cfg

This is an object of type `kiss_fftr_state` (which is a C data structure).

kiss_fftr_state

The `kiss_fftr_state` is a data type which defines the required variables for the Fourier transform and allocates the required space. This has the same purpose as `kiss_fft_state` but for the Fourier transform of the real signals.

sf_complex

This is an object of type `kiss_fft_cfg` (which is an object of C data structure).

sf_double_complex

This is a C data structure for complex numbers. It uses the type double for the real and imaginary parts of the complex numbers.

3.1.2 Files

This section lists the data types used to define the `.rsf` file structure.

sf_file

This is an object of type `sf_File`. `sf_File` is a data structure which defines the variables required for creating a `.rsf` file in Madagascar. It is defined in `file.c`.

sf_datatype

This is a C enumeration, which means that it contains new data types, which are not the fundamental types like int, float, `sf_file` etc. This data type is used in `sf_File` data structure to set the type of a `.rsf` file, for example SF_CHAR, SF_INT etc. It is defined in `file.c`.

sf_dataform

This is a C enumeration, which means that it contains new data types, which are not the fundamental types like int, float, `sf_file` etc. This data type is used in `sf_File` data structure to set the format of an `.rsf` file, for example `SF_ASCII`, `SF_XDR` and `SF_NATIVE`. It is defined in `file.c`.

3.1.3 Operators

This section lists the data types used to define linear operators.

sf_triangle

This is an object of an abstract C datatype type `sf_Triangle`. The `sf_triangle` data type defines the variables of relevant types to store information about the triangle smoothing filter. It is defined in `triangle.c`.

sf_operator

This is a C data type of type void. It is also a pointer to a function which takes the input parameters precisely as `(bool, bool, int, int, float*, float*)`. It is defined in `_solver.h`.

sf_solverstep

This is a C data type of type void. It is also a pointer to a function which takes the input parameters precisely as `(bool, bool, int, int, float*, const float*, float*, const float*)`. It is defined in `_solver.h`.

sf_weight

This is a C data type of type void. It is also a pointer to a function which takes the input parameters precisely as `(int, int, const sf_complex*, float*)`. It is defined in `_solver.h`.

sf_coperator

This is a C data type of type void. It is also a pointer to a function which takes the input parameters precisely as `(bool bool, int, int, sf_complex*, sf_complex*)`. It works just like `sf_operator` but does it for complex numbers. It is defined in `_solver.h`.

sf_csolverstep

This is a C data type of type void. It is also a pointer to a function which takes the input parameters precisely as `(bool, bool, int, int, sf_complex*, const sf_complex*, sf_complex*, const sf_complex*)`. It works just like `sf_solverstep` but does it for complex numbers. It is defined in `_solver.h`.

sf_cweight

This is a C data type of type void. It is also a pointer to a function which takes the input parameters precisely as `(int, int, const sf_complex*, float*)`. It works just like `sf_weight` but does it for the complex numbers. It is defined in `_solver.h`.

sf_eno

This is a C data structure, which contains the required variables for 1D ENO (Essentially Non Oscillatory) interpolation. It is defined in `eno.c`.

sf_eno2

This is a C data structure, which contains the required variables for 2D ENO (Essentially Non Oscillatory) interpolation. It is defined in `eno2.c`.

sf_bands

This is a C data structure, which contains the required variables for storing a banded matrix. It is defined in `banded.c`.

3.1.4 Geometry

This section lists the data types used to define the geometry of the seismic data.

sf_axa

This is a C data structure which contains the variables of type int and float to store the length origin and sampling of the axis. It is defined in `axa.c`.

pt2d

This is a C data structure which contains the variables of type double and float to store the location and value of a 2D point. It is defined in `point.c`.

pt3d

This is a C data structure which contains the variables of type double and float to store the location and value of a 3D point. It is defined in `point.c`.

vc2d

This is a C data structure which contains the variables of type double to store the components of a 2D vector. It is defined in `vector.c`.

vc3d

This is a C data structure which contains the variables of type double to store the components of a 3D vector. It is defined in `vector.c`.

3.1.5 Lists

This section describes the data types used to create and operate on lists.

sf_list

This is a C data structure, which contains the required variables for storing the information about the list, for example . It uses another C data structure Entry. It is defined in `llist.c`.

Entry

This is a C data structure, which contains the required variables for storing the elements and moving the pointer in the list. It is defined in `llist.c`.

3.1.6 sys/types.h

This section describes some of the data types used from the C header file `sys/types.h`.

off_t

This is a data type defined in the `sys/types.h` header file (of fundamental type `unsigned long`) and is used to measure the file offset in bytes from the beginning of the file. It is defined as a signed, 32-bit integer, but if the programming environment enables large files `off_t` is defined to be a signed, 64-bit integer.

size_t

This is a data type defined in the `sys/types.h` header (of fundamental type `unsigned int`) and is used to measure the file size in units of character. It is used to hold the result of the sizeof operator in C, for example sizeof(int)=4, sizeof(char)=1, etc.

Chapter 4

Preparing for input

4.1 Convenience allocation programs (alloc.c)

4.1.1 sf_alloc

Checks whether the requested size for memory allocation is valid and if so it returns a pointer of void type, pointing to the allocated memory block. It takes the 'number of elements' and 'size of one element' as input arguments. Both arguments have to be of the of type `size_t`.

Call

```
sf_alloc (n, size);
```

Definition

```
void *sf_alloc (size_t n    /* number of elements */,
                size_t size /* size of one element */)
          /*< output-checking allocation >*/
{
   ...
}
```

Input parameters

`n`	number of elements (`size_t`).
`size`	size of each element, for example `sizeof(float)` (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.2 sf_realloc

The same as `sf_alloc` but it allocates new memory such that it appends the block previously assigned by `sf_alloc`. It takes three parameters, first one is a void pointer to the old memory block. Second and third parameters are the same as for `sf_alloc` but are used to determine the new block, which is to be appended. `sf_realloc` returns a void pointer pointing to the whole memory block (new + old).

Call

```
sf_realloc (ptr, n, size);
```

Definition

```
void *sf_realloc (void* ptr   /* previous data */,
                  size_t n    /* number of elements */,
                  size_t size /* size of one element */)
/*< output-checking reallocation >*/
{
   ...
}
```

Input parameters

`ptr`	pointer to the previously assigned memory block.
`n`	number of elements (`size_t`).
`size`	size of each element, for example `sizeof(float)` (`size_t`).

Output

`ptr`	pointer to the new aggregate block.

4.1.3 sf_charalloc

Allocates the memory exactly like `sf_alloc` but the size in this one is fixed which is the size of one character. Therefore `sf_charalloc` allocates the memory for `n` elements which must be of character type. Because the size is fixed there is just one input parameter which is the number of elements (i.e. characters). Output is a void pointer pointing to the block of memory allocated.

Call

```
ptr = sf_charalloc (n);
```

Definition

```
char *sf_charalloc (size_t n /* number of elements */)
/*< char allocation >*/
{
   ...
}
```

Input parameters

`n`	number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.4 sf_ucharalloc

The same as `sf_charalloc` but it only allocates the memory for the unsigned character type, that is, the size of the elements is `sizeof(unsigned char)`.

Call

```
ptr = sf_ucharalloc (n);
```

Definition

```
unsigned char *sf_ucharalloc (size_t n /* number of elements */)
/*< unsigned char allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.5 sf_shortalloc

Allocates the memory for the short integer type, that is, the size of the elements is, for example `sizeof(short int)`.

Call

```
ptr = sf_shortalloc (n);
```

Definition

```
short *sf_shortalloc (size_t n /* number of elements */)
/*< short allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.6 sf_intalloc

Allocates the memory for the large integer type, that is, the size of the elements is, for example `sizeof(int)`.

Call

```
ptr = sf_intalloc (n);
```

Definition

```
int *sf_intalloc (size_t n /* number of elements */)
          /*< int allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.7 sf_largeintalloc

Allocates the memory for the large integer type, that is, the size of the elements is, for example `sizeof(large int)`.

Call

```
ptr = sf_largeintalloc (n);
```

Definition

```
off_t *sf_largeintalloc (size_t n /* number of elements */)
/*< sf_largeint allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.8 sf_floatalloc

Allocates the memory for the floating point type, that is, the size of the elements is, for example `sizeof(float)`.

Call

```
ptr = sf_floatalloc (n);
```

Definition

```
float *sf_floatalloc (size_t n /* number of elements */)
          /*< float allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.9 sf_complexalloc

Allocates the memory for the `sf_complex` type, that is, the size of the elements is, for example `sizeof(sf_complex)`.

Call

```
ptr = sf_complexalloc (n);
```

Definition

```
sf_complex *sf_complexalloc (size_t n /* number of elements */)
/*< complex allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.10 sf_complexalloc2

Allocates a 2D array in the memory for the `sf_complex` type. It works just like `sf_complexalloc` but does it for two dimensions. This is done by making a pointer point to another pointer, which in turn points to a particular column (or row) of an allocated 2D block of memory of size `n1*n2`. `n1` is the fastest dimension.

Call

```
ptr = sf_complexalloc2 (n1, n2);
```

Definition

```
sf_complex **sf_complexalloc2 (size_t n1 /* fast dimension */,
                               size_t n2 /* slow dimension */)
/*< complex 2-D allocation, out[0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.11 sf_complexalloc3

Allocates a 3D array in the memory for the `sf_complex` type. It works just like `sf_complexalloc2` but does it for three dimensions. This is done by extending the same argument as for `sf_complexalloc2` this time making a pointer such that `Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_complexalloc3 (n1, n2, n3);
```

Definition

```
sf_complex ***sf_complexalloc3 (size_t n1 /* fast dimension */,
                                size_t n2 /* slower dimension */,
                                size_t n3 /* slowest dimension */)
/*< complex 3-D allocation, out[0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.12 sf_complexalloc4

Allocates a 4D array in the memory for the `sf_complex` type. It works just like `sf_complexalloc2` but does it for four dimensions. This is done by extending the same argument as for `sf_complexalloc2` but this time making a pointer such that `Pointer3 -> Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_complexalloc4 (n1, n2, n3, n4);
```

Definition

```
sf_complex ****sf_complexalloc4 (size_t n1 /* fast dimension */,
                                 size_t n2 /* slower dimension */,
                                 size_t n3 /* slower dimension */,
                                 size_t n4 /* slowest dimension */)
/*< complex 4-D allocation, out[0][0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).
`n4`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.13 sf_boolalloc

Allocates the memory for the bool type, that is, the size of the elements is, for example `sizeof(bool)`.

Call

```
ptr = sf_boolalloc (n);
```

Definition

```
bool *sf_boolalloc (size_t n /* number of elements */)
/*< bool allocation >*/
{
   ...
}
```

Input parameters

`n` number of elements (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.14 sf_boolalloc2

Allocates a 2D array in the memory for the bool type. It works just like `sf_boolalloc` but does it for two dimensions. This is done by making a pointer point to another pointer, which in turn points to a particular column (or row) of an allocated 2D block of memory of size `n1*n2`. `n1` is the fastest dimension.

Call

```
ptr = sf_boolalloc2 (n1, n2);
```

Definition

```
bool **sf_boolalloc2 (size_t n1 /* fast dimension */,
                      size_t n2 /* slow dimension */)
/*< bool 2-D allocation, out[0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1` number of elements in the fastest dimension (`size_t`).

`n2` number of elements in the slower dimension (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.15 sf_boolalloc3

Allocates a 3D array in the memory for the bool type. It works just like `sf_boolalloc2` but does it for three dimensions. This is done by extending the same argument as for `sf_boolalloc2` but this time making a pointer such that `Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_boolalloc3 (n1, n2, n3);
```

Definition

```
bool ***sf_boolalloc3 (size_t n1 /* fast dimension */,
                       size_t n2 /* slower dimension */,
                       size_t n3 /* slowest dimension */)
/*< bool 3-D allocation, out[0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.16 sf_floatalloc2

Allocates a 2D array in the memory for the float type. It works just like `sf_floatalloc` but does it for two dimensions. This is done by making a pointer point to another pointer, which in turn points to a particular column (or row) of an allocated 2D block of memory of size `n1*n2`. `n1` is the fastest dimension.

Call

```
ptr = sf_floatalloc2 (n1, n2);
```

Definition

```
float **sf_floatalloc2 (size_t n1 /* fast dimension */,
                        size_t n2 /* slow dimension */)
/*< float 2-D allocation, out[0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

n1 number of elements in the fastest dimension (size_t).

n2 number of elements in the slower dimension (size_t).

Output

ptr a void pointer pointing to the allocated block of memory.

4.1.17 sf_floatalloc3

Allocates a 3D array in the memory for the float type. It works just like sf_floatalloc2 but does it for three dimensions. This is done by extending the same argument as for sf_floatalloc2 but this time making a pointer such that Pointer2 -> Pointer1 -> Pointer. n1 is the fastest dimension.

Call

```
ptr = sf_floatalloc3 (n1, n2, n3);
```

Definition

```
float ***sf_floatalloc3 (size_t n1 /* fast dimension */,
                         size_t n2 /* slower dimension */,
                         size_t n3 /* slowest dimension */)
/*< float 3-D allocation, out[0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

n1 number of elements in the fastest dimension (size_t).

n2 number of elements in the slower dimension (size_t).

n3 number of elements in the slower dimension (size_t).

Output

ptr a void pointer pointing to the allocated block of memory.

4.1.18 sf_floatalloc4

Allocates a 4D array in the memory for the float type. It works just like sf_floatalloc2 but does it for four dimensions. This is done by extending the same argument as for sf_floatalloc2 but this time making a pointer such that Pointer3 -> Pointer2 -> Pointer1 -> Pointer. n1 is the fastest dimension.

Call

```
ptr = sf_floatalloc4 (n1, n2, n3, n4);
```

Definition

```
float ****sf_floatalloc4 (size_t n1 /* fast dimension */,
                          size_t n2 /* slower dimension */,
                          size_t n3 /* slower dimension */,
                          size_t n4 /* slowest dimension */)
/*< float 4-D allocation, out[0][0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

n1	number of elements in the fastest dimension (size_t).
n2	number of elements in the slower dimension (size_t).
n3	number of elements in the slower dimension (size_t).
n4	number of elements in the slower dimension (size_t).

Output

ptr	a void pointer pointing to the allocated block of memory.

4.1.19 sf_floatalloc5

Allocates a 5D array in the memory for the float type. It works just like `sf_floatalloc2` but does it for four dimensions. This is done by extending the same argument as for `sf_floatalloc2` but this time making a pointer such that `Pointer4 -> Pointer3 -> Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_floatalloc5 (n1, n2, n3, n4, n5);
```

Definition

```
float *****sf_floatalloc5 (size_t n1 /* fast dimension */,
                           size_t n2 /* slower dimension */,
                           size_t n3 /* slower dimension */,
                           size_t n4 /* slower dimension */,
                           size_t n5 /* slowest dimension */)
/*< float 5-D allocation, out[0][0][0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

n1	number of elements in the fastest dimension (size_t).
n2	number of elements in the slower dimension (size_t).
n3	number of elements in the slower dimension (size_t).

`n4`	number of elements in the slower dimension (`size_t`).
`n5`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.20 sf_floatalloc6

Allocates a 6D array in the memory for the float type. It works just like `sf_floatalloc2` but does it for four dimensions. This is done by extending the same argument as for `sf_floatalloc2` but this time making a pointer such that `Pointer5 -> Pointer4 -> Pointer3 -> Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_floatalloc6 (n1, n2, n3, n4, n5, n6);
```

Definition

```
float ******sf_floatalloc6 (size_t n1 /* fast dimension */,
                            size_t n2 /* slower dimension */,
                            size_t n3 /* slower dimension */,
                            size_t n4 /* slower dimension */,
                            size_t n5 /* slower dimension */,
                            size_t n6 /* slowest dimension */)
/*< float 6-D allocation, out[0][0][0][0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).
`n4`	number of elements in the slower dimension (`size_t`).
`n5`	number of elements in the slower dimension (`size_t`).
`n6`	number of elements in the slower dimension. Must be of type `size_t`

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.21 sf_intalloc2

Allocates a 2D array in the memory for the float type. It works just like `sf_intalloc` but does it for two dimensions. This is done by making a pointer point to another pointer, which in turn points to a particular column (or row) of an allocated 2D block of memory of size `n1*n2`. `n1` is the fastest dimension.

Call

```
ptr = sf_intalloc2 (n1, n2);
```

Definition

```
int **sf_intalloc2 (size_t n1 /* fast dimension */,
                    size_t n2 /* slow dimension */)
/*< float 2-D allocation, out[0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.22 sf_intalloc3

Allocates a 3D array in the memory for the float type. It works just like `sf_intalloc2` but does it for three dimensions. This is done by extending the same argument as for `sf_intalloc2` this time making a pointer such that `Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_intalloc3 (n1, n2, n3);
```

Definition

```
int ***sf_intalloc3 (size_t n1 /* fast dimension */,
                     size_t n2 /* slower dimension */,
                     size_t n3 /* slowest dimension */)
/*< int 3-D allocation, out[0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.23 sf_intalloc4

Allocates a 4D array in the memory for the float type. It works just like `sf_intalloc2` but does it for four dimensions. This is done by extending the same argument as for `sf_intalloc2` but this time making a pointer such that `Pointer3 -> Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_intalloc4 (n1, n2, n3, n4);
```

Definition

```
int ****sf_intalloc4 (size_t n1 /* fast dimension */,
                      size_t n2 /* slower dimension */,
                      size_t n3 /* slower dimension */,
                      size_t n4 /* slowest dimension */ )
/*< int 4-D allocation, out[0][0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).
`n4`	number of elements in the slower dimension (`size_t`).

Output

`ptr` a void pointer pointing to the allocated block of memory.

4.1.24 sf_charalloc2

Allocates a 2D array in the memory for the float type. It works just like `sf_charalloc` but does it for two dimensions. This is done by making a pointer point to another pointer, which in turn points to a particular column (or row) of an allocated 2D block of memory of size `n1*n2`. `n1` is the fastest dimension.

Call

```
ptr = sf_charalloc2 (n1, n2);
```

Definition

```
char **sf_charalloc2 (size_t n1 /* fast dimension */,
                      size_t n2 /* slow dimension */)
/*< char 2-D allocation, out[0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.25 sf_uncharalloc2

Allocates a 2D array in the memory for the float type. It works just like `sf_uncharalloc` but does it for two dimensions. This is done by making a pointer point to another pointer, which in turn points to a particular column (or row) of an allocated 2D block of memory of size `n1*n2`. `n1` is the fastest dimension.

Call

```
ptr = sf_ucharalloc2 (n1, n2);
```

Definition

```
unsigned char **sf_ucharalloc2 (size_t n1 /* fast dimension */,
                                size_t n2 /* slow dimension */)
/*< unsigned char 2-D allocation, out[0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.1.26 sf_uncharalloc3

Allocates a 3D array in the memory for the float type. It works just like `sf_uncharalloc2` but does it for three dimensions. This is done by extending the same argument as for `sf_uncharalloc2` but this time making a pointer such that `Pointer2 -> Pointer1 -> Pointer`. `n1` is the fastest dimension.

Call

```
ptr = sf_ucharalloc3 (n1, n2, n3);
```

Definition

```
unsigned char ***sf_ucharalloc3 (size_t n1 /* fast dimension */,
                                 size_t n2 /* slower dimension */,
                                 size_t n3 /* slowest dimension */)
/*< unsigned char 3-D allocation, out[0][0] points to a contiguous array >*/
{
   ...
}
```

Input parameters

`n1`	number of elements in the fastest dimension (`size_t`).
`n2`	number of elements in the slower dimension (`size_t`).
`n3`	number of elements in the slower dimension (`size_t`).

Output

`ptr`	a void pointer pointing to the allocated block of memory.

4.2 Simbol Table for parameters (simtab.c)

4.2.1 sf_simtab_init

Creates a table to store the parameters input either from command line or a file. It takes the required size (type `int`) of the table as input. The output is a pointer to the allocated table and it is of the defined data type `sf_simtab`.

call

```
table = sf_simtab_init(size);
```

Definition

```
sf_simtab sf_simtab_init(int size)
/*< Create simbol table. >*/
{
   ...
}
```

Input parameters

`size`	size of the table to be allocated (`int`).

Output

`table`	a pointer of type `sf_simtab` pointing to the allocated block of memory for the symbol table.

4.2.2 sf_simtab_close

Frees the allocated space for the table.

Call

```
sf_simtab_close(table);
```

Definition

```
void sf_simtab_close(sf_simtab table)
/*< Free allocated memory >*/
{
   ...
}
```

Input parameters

`table`	the table whose allocated memory has to be deleted. Must be of type `sf_simtab`.

4.2.3 sf_simtab_enter

Enters a value in the table, which was created by `sf_simtab_init`. In the input it must be told which table to enter the value in, this is the first input argument and is of type `sf_simtab`. The second and the third arguments are the pointers of `const char*` type. The first one points to `key`, which would be the name of the argument from command line or file. Second argument is the pointer to the value to be input.

Call

```
sf_simtab_enter(table, key, val);
```

Definition

```
void sf_simtab_enter(sf_simtab table, const char *key, const char* val)
/*< Add an entry key=val to the table >*/
{
    ...
}
```

Input parameters

`table`	the table in which the the key value is to be stored. Must be of type `sf_simtab`.
`key`	pointer to the name of the key value to be input (`const char*`).
`val`	pointer to the key value to be input (`const char*`).

4.2.4 sf_simtab_get

Extracts the value of the input key from the symbol table. It is used in other functions such as `sf_simtab_getint`.

Call

```
val = sf_simtab_get(table, key);
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).

Output

`val`	pointer of type `char` to the desired key value stored in the table. This is the output in case there is a match between the required key and a key in the table. If there is no match between the required key and the key stored in the table, then `NULL` is returned.

4.2.5 sf_simtab_getint

Extracts an integer from the table. If the extraction is successful returns a boolean true, otherwise returns a false.

Call

```
success = sf_simtab_getint (table, key, par);
```

Definition

```
bool sf_simtab_getint (sf_simtab table, const char* key,/*@out@*/ int* par)
/*< extract an int parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the integer variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.6 sf_simtab_getlargeint

Extracts a large integer from the table. If the extraction is successful, it returns a boolean true, otherwise a false.

Call

```
success = sf_simtab_getlargeint (table, key, par);
```

Definition

```
bool sf_simtab_getlargeint (sf_simtab table, const char* key,/*@out@*/ off_t* pa
r)
/*< extract a sf_largeint parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the large integer variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.7 sf_simtab_getfloat

Extracts a float value from the table. If the extraction is successful, it returns a boolean true, otherwise a false.

Call

```
success = sf_simtab_getfloat (table, key, par);
```

Definition

```
bool sf_simtab_getfloat (sf_simtab table, const char* key,/*@out@*/ float* par)
/*< extract a float parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the float type value variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise successfully.

4.2.8 sf_simtab_getdouble

Extracts a double type value from the table. If the extraction is successful, it returns a boolean true, otherwise a false.

Call

```
success = sf_simtab_getdouble (table, key, par);
```

Definition

```
bool sf_simtab_getdouble (sf_simtab table, const char* key,/*@out@*/ double* par
)
/*< extract a double parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the double type value variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.9 sf_simtab_getfloats

Extracts an array of float values from the table. If the extraction is successful, it returns a boolean `true`, otherwise a `false`.

Call

```
 success = sf_simtab_getfloats (table, key, par, n);
```

Definition

```
bool sf_simtab_getfloats (sf_simtab table, const char* key,
                          /*@out@*/ float* par,size_t n)
/*< extract a float array parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the `float` array which has to be extracted (`const char*`).
`par`	pointer to the array of `float` type value variable where the extracted value id to be copied.
`n`	size of the array to be extracted (`size_t`).

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.10 sf_simtab_getstring

Extracts a string pointed by the input key from the symbol table. If the value is `NULL` it will return `NULL`, otherwise it will allocate a new block of memory of char type and copy the memory block from the table to the new block and return a pointer to the newly allocated block of memory.

Call

```
string = sf_simtab_getstring (table, key);
```

Definition

```
char* sf_simtab_getstring (sf_simtab table, const char* key)
/*< extract a string parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the string has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the string which has to be extracted (`const char*`).

Output

`string`	a pointer to allocated block of memory containing a string of characters.

4.2.11 sf_simtab_getbool

Extracts a boolean value from the table. If the extraction is successful, it returns a boolean `true`, otherwise a `false`.

Call

```
success = sf_simtab_getbool (table, key, par);
```

Definition

```
bool sf_simtab_getbool (sf_simtab table, const char* key,/*@out@*/ bool *par)
/*< extract a bool parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the value has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the bool variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.12 sf_simtab_getbools

Extracts an array of boolean values from the table. If the extraction is successful, it returns a boolean `true`, otherwise a `false`.

Call

```
success = sf_simtab_getbools (table, key, par, n);
```

Definition

```
sf_simtab_getbools (sf_simtab table, const char* key,/*@out@*/bool *par,size_t n)
/*< extract a bool array parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the boolean array which has to be extracted. Must be of XXXXXXXXX pointer to the array of `bool` type value variable where the extracted value is to be copied.
`n`	size of the array to be extracted (`size_t`).

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.13 sf_simtab_getints

Extracts an array of integer values from the table. If the extraction is successful, it returns a boolean `true`, otherwise a `false`.

Call

```
success = sf_simtab_getints (table, key, par, n);
```

Definition

```
bool sf_simtab_getints (sf_simtab table, const char* key,
                        /*@out@*/ int *par,size_t n)
/*< extract an int array parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the integer array which has to be extracted (`const char*`).
`par`	pointer to the array of integer type value variable where the extracted value id to be copied.
`n`	size of the array to be extracted. Must be of `size_t`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.14 sf_simtab_getstrings

Extracts an array of strings from the table. is successful, it returns a boolean `true`, otherwise a `false`.

Call

```
success = sf_simtab_getstrings (table, key, par, n);
```

Definition

```
bool sf_simtab_getstrings (sf_simtab table, const char* key,
                           /*@out@*/ char **par,size_t n)
/*< extract a string array parameter from the table >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the string array which has to be extracted (`const char*`).
`par`	pointer to the pointer to array of integer type value variable where the extracted value is to be copied.
`n`	size of the array to be extracted. Must be of `size_t`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.2.15 sf_simtab_put

Writes a new key together with its value to the symbol table. The new entry must be in the form `key=val` and must be of the `const char*` type, that is, this function must be given a pointer to `key=val`. Since the type of the pointer is `const char*` this can be a direct input from the command line and in that case the pointer will be `acgv[n]` where `n` specifies the position in the command line.

Call

```
sf_simtab_put (table, keyval);
```

Definition

```
void sf_simtab_put (sf_simtab table, const char *keyval)
/*< put a key=val string to the table >*/
{
   ...
}
```

Input parameters

`table`	the table in which the value has to be entered. Must be of type `sf_simtab`.
`keyval`	pointer to `key=val` which is to be entered.

4.2.16 sf_simtab_input

Inputs a table from one file and copies it into another and also adds the new entry into the internal table using `sf_simtab_put`.

Call

```
sf_simtab_input ( table, fp, out);
```

Definition

```
void sf_simtab_input (sf_simtab table, FILE* fp, FILE* out)
/*< extract parameters from a file >*/
{
   ...
}
```

Input parameters

`table`	the table in which the value has to be entered. Must be of type `sf_simtab`.
`fp`	pointer to the file from which the parameter is to be read. It must be of type`FILE*`.
`out`	pointer to the file in which the parameter is to be written. It must be of type FILE*.

4.2.17 sf_simtab_output

Reads the parameters from the internal table and writes them to a file.

Call

```
sf_simtab_output ( table, fp);
```

Definition

```
void sf_simtab_output (sf_simtab table, FILE* fp)
/*< output parameters to a file >*/
{
   ...
}
```

Input parameters

`table`	the table in which the value has to be entered. Must be of type `sf_simtab`.
`fp`	pointer to the file in which the parameter is to be written. It must be of type `FILE*`.

4.3 Parameter handling (getpar.c)

4.3.1 sf_stdin

Checks whether there is an input in the command line, if not it returns a false. It reads the first character in the file: if it is an `EOF`, `false` is returned and if not, then `true` is the return value. It takes no input parameters and returns a boolean value.

Call

```
hasinp sf_stdin();
```

Definition

```
bool sf_stdin(void)
/*< returns true if there is an input in stdin >*/
{
   ...
}
```

4.3.2 sf_init

Initializes a parameter table which is created using `sf_simtab_init`. The input arguments are same as used in the main function in c when there is some arguments are to be input from the command line.

Input parameters

`argc`	size of the table to be allocated (`int`).
`argv[]`	pointer to the character array from the command line (`char*`).

Call

```
sf_init(argc, argv[]);
```

Definition

```
void sf_init(int argc,char *argv[])
/*< initialize parameter table from command-line arguments >*/
{
   ...
}
```

4.3.3 sf_par_close

Frees the allocated space for the table. It uses `sf_simtab_close` to close the table. It does not take any input parameters but passes a pointer `pars` defined by `sf_init`.

Call

```
sf_parclose ();
```

Definition

```
void sf_parclose (void)
/*< close parameter table and free space >*/
{
   ...
}
```

4.3.4 sf_parout

Reads the parameters from the internal table and writes them to a file. It uses `sf_simtab_output`. It takes a pointer to the file, in which the parameters are to be written.

Call

```
sf_parout (file);
```

Definition

```
void sf_parout (FILE *file)
/*< write the parameters to a file >*/
{
   ...
}
```

Input parameters

`parout`	the table in which the value has to be entered (`FILE*`).

4.3.5 sf_getprog

Outputs a pointer of `char` type to the name of the current running program. The pointer it returns is assigned a value in `sf_init`.

Call

```
prog = sf_getprog ();
```

Definition

```
char* sf_getprog (void)
/*< returns name of the running program >*/
{
   ...
}
```

Output

`prog`	pointer to an array which contains the current program name (`char`).

4.3.6 sf_getuser

Outputs a pointer of `char` type to the name of the current user. The pointer it returns is assigned a value in `sf_init`.

Call

```
user = sf_getuser ();
```

Definition

```
char* sf_getuser (void)
/*< returns user name >*/
{
   ...
}
```

Output

`user` pointer to an array which contains the user name (`char`).

4.3.7 sf_gethost

Outputs a pointer of `char` type to the name of the current host. The pointer it returns is assigned a value in `sf_init`.

Call

```
host = sf_gethost ();
```

Definition

```
char* sf_gethost (void)
/*< returns host name >*/
{
   ...
}
```

Output

`host` pointer to an array which contains the host name (`char`).

4.3.8 sf_getcdir

Outputs a pointer of `char` type to the name of the current working directory. The pointer it returns is assigned a value in `sf_init`.

Call

```
cdir = sf_getcdir ();
```

Definition

```
char* sf_getcdir (void)
/*< returns current directory >*/
{
   ...
}
```

Output

`cdir` pointer to an array which contains the current working directory (`char`).

4.3.9 sf_getint

Extracts an integer from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`. It uses `sf_simtab_getint`.

Call

```
success = sf_getint (key, par);
```

Definition

```
bool sf_getint (const char* key,/*@out@*/ int* par)
/*< get an int parameter from the command line >*/
{
   ...
}
```

Input parameters

`key` the name of the entry which has to be extracted (`const char*`).

`par` pointer to the integer variable where the extracted value is to be copied.

Output

`success` a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.10 sf_getlargeint

Extracts a large integer from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`. It uses `sf_simtab_getlargeint`.

Call

```
sf_getlargeint (key, par);
```

Definition

```
bool sf_getlargeint (const char* key,/*@out@*/ off_t* par)
/*< get a large int parameter from the command line >*/
{
   ...
}
```

Input parameters

`key` the name of the entry which has to be extracted (`const char*`).

`par` pointer to the large integer variable where the extracted value is to be copied. Must be of type `off_t` which is defined in the header `<sys/types.h>`.

Output

`success` a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.11 sf_getints

Extracts an array of integer values from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_getints (key, par, n);
```

Definition

```
bool sf_getints (const char* key,/*@out@*/ int* par,size_t n)
/*< get an int array parameter (comma-separated) from the command line >*/
{
   ...
}
```

Input parameters

`key` the name of the integer array which has to be extracted (`const char*`).

`par` pointer to the array of integer type value variable where the extracted value id to be copied.

`n` size of the array to be extracted. Must be of `size_t`.

Output

`success` a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.12 sf_getfloat

Extracts a float value from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_getfloat (key, par);
```

Definition

```
bool sf_getfloat (const char* key,/*@out@*/ float* par)
/*< get a float parameter from the command line >*/
{
   ...
}
```

Input parameters

`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the float type value variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.13 sf_getdouble

Extracts a double type value from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_getdouble (key, par);
```

Definition

```
bool sf_getdouble (const char* key,/*@out@*/ double* par)
/*< get a double parameter from the command line >*/
{
   ...
}
```

Input parameters

`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the double type value variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.14 sf_getfloats

Extracts an array of float values from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`. It uses `sf_simtab_getfloats`.

Call

```
success = sf_getfloats (key, n);
```

Definition

```
bool sf_getfloats (const char* key,/*@out@*/ float* par,size_t n)
/*< get a float array parameter from the command line >*/
{
   ...
}
```

Input parameters

`key`	the name of the float array which has to be extracted (`const char*`).
`par`	pointer to the array of float type value variable where the extracted value id to be copied.
`n`	size of the array to be extracted (`size_t`).

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.15 sf_getstring

Extracts a string pointed by the input key from the command line. If the value is `NULL` it will return `NULL`, otherwise it will allocate a new block of memory of `char` type and copy the memory block from the table to the new block and return a pointer to the newly allocated block of memory. It uses `sf_simtab_getstring`.

Call

```
string = sf_getstring (key);
```

Definition

```
char* sf_getstring (const char* key)
/*< get a string parameter from the command line >*/
{
   ...
}
```

Input parameters

`key`	the name of the string which has to be extracted (`const char*`).

Output

`string`	a pointer to allocated block of memory containing a string of characters.

4.3.16 sf_getstrings

Extracts an array of strings from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_getstrings (key, par, n);
```

Definition

```
bool sf_getstrings (const char* key,/*@out@*/ char** par,size_t n)
/*< get a string array parameter from the command line >*/
{
   ...
}
```

Input parameters

`key`	the name of the string array which has to be extracted (`const char*`).
`par`	pointer to the pointer to array of integer type value variable where the extracted value is to be copied.
`n`	size of the array to be extracted. Must be of `size_t`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.17 sf_getbool

Extracts a boolean value from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_getbool (key, par);
```

Definition

```
bool sf_getbool (const char* key,/*@out@*/ bool* par)
/*< get a bool parameter from the command line >*/
{
   ...
}
```

Input parameters

`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the bool variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.18 sf_getbools

Extracts an array of bool values from the command line. If the extraction is successful, it returns a `true`, otherwise a `false`. It uses `sf_simtab_getbools`.

Call

```
success = sf_getbools (key, par, n);
```

Definition

```
bool sf_getbools (const char* key,/*@out@*/ bool* par,size_t n)
/*< get a bool array parameter from the command line >*/
{
    return sf_simtab_getbools(pars,key,par,n);
}
```

Input parameters

`key`	the name of the bool array which has to be extracted (`const char*`).
`par`	pointer to the array of bool type value variable where the extracted value id to be copied.
`n`	size of the array to be extracted. Must be of `size_t`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

4.3.19 sf_getpars

This function returns a pointer to the parameter table which must have been initialized earlier in the program using `sf_init`.

Call

```
pars = sf_getpars (void);
```

Definition

```
sf_simtab sf_getpars (void)
/*< provide access to the parameter table >*/
{
   ...
}
```

Output

`pars`	a pointer to the parameter table.

Chapter 5

Operations with RSF files

5.1 Main operations with RSF files (file.c)

5.1.1 sf_file_error

Sets an error on opening files. It sets the value of `error` (a static variable of type `bool`). This variable is used in the `sf_input_error` as an if-condition.

Call

```
sf_file_error(err);
```

Definition

```
void sf_file_error(bool err)
/*< set error on opening files >*/
{
   ...
}
```

Input parameters

`err`	value of type `bool` which is to be assigned to the static variable `error`.

5.1.2 sf_error

Outputs an error message to the `stderr` (usually the screen) if the file cannot be opened. The ':' after the format specifiers in the call to `sf_error` ensures that any system errors are also included in the output.

Call

```
sf_input_error(file, message, name);
```

Definition

```
static void sf_input_error(sf_file file, const char* message, const char* name)
{
   ...
}
```

Input parameters

`file`	pointer to the input file structure (`sf_file`).
`message`	the error message to be output to `stderr`.
`name`	name of the file which was to be opened.

5.1.3 sf_input

Creates an input file structure and returns a pointer to that file structure. It will create the symbol table, input parameters to the table and write them in a temporary file and check for any errors in the input of the parameters. It will also set the format of the file and then return a pointer to the file structure.

Call

```
file = sf_input(tag);
```

Definition

```
sf_file sf_input (/*@null@*/ const char* tag)
/*< Create an input file structure >*/
{
   ...
}
```

Input parameters

`tag`	a tag for the input file (`const char*`).

Output

`file`	a pointer to the input file structure.

5.1.4 sf_output

Creates an output file structure and returns a pointer to that file structure. It will create the symbol table, a header file and put the path of the data file in the header with the key "`in`".

Call

```
file = sf_output(tag);
```

Definition

```
sf_file sf_output (/*@null@*/ const char* tag)
/*< Create an output file structure.
---
Should do output after sf_input. >*/
{
   ...
}
```

Input parameters

`tag` a tag for the output file (`const char*`).

Output

`file` a pointer to the input file structure.

5.1.5 sf_gettype

Returns the type of the file, e.g. `SF_INT`, `SF_FLOAT`, `SF_COMPLEX` etc.

Call

```
type = sf_gettype (file);
```

Definition

```
sf_datatype sf_gettype (sf_file file)
/*< return file type >*/
{
   ...
}
```

Input parameters

`file` a pointer to the file structure whose type is required (`sf_file`).

Output

`file->type` type of the file structure.

5.1.6 sf_getform

Returns the file form, e.g. `SF_ASCII`, `SF_XDR`.

Call

```
form = sf_getform (file);
```

Definition

```
sf_dataform sf_getform (sf_file file)
/*< return file form >*/
{
   ...
}
```

Input parameters

`file` a pointer to the file structure whose type is required (`sf_file`).

Output

`file->form` form of the file structure.

5.1.7 sf_esize

Returns the size of the element type of the file, e.g. `SF_INT`, `SF_FLOAT`, `SF_COMPLEX` etc.

Call

```
size = sf_esize(file);
```

Definition

```
size_t sf_esize(sf_file file)
/*< return element size >*/
{
   ...
}
```

Input parameters

`file` a pointer to the file structure whose type is required (`sf_file`).

Output

`size` size in bytes of the type of the file structure.

5.1.8 sf_settype

Sets the type of the file, e.g. `SF_INT`, `SF_FLOAT`, `SF_COMPLEX` etc.

Call

```
sf_settype (file,type);
```

Definition

```
void sf_settype (sf_file file, sf_datatype type)
/*< set file type >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose type is to be set (`sf_file`).
`type`	the type to be set. Must be of type `sf_datatype`, e.g. `SF_INT`.

5.1.9 sf_setpars

Changes the parameter table from that of the file to the one which has parameters from the command line.

Call

```
sf_setpars (file);
```

Definition

```
void sf_setpars (sf_file file)
/*< change parameters to those from the command line >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose parameter table is to be closed (`sf_file`).

5.1.10 sf_bufsiz

Returns the size of the buffer associated with the file. It gets the buffer size using the file descriptor of the file and the predefined structure `stat`. This provides control over the I/O operations, making them more efficient.

Call

```
bufsiz = sf_bufsiz(file);
```

Definition

```
size_t sf_bufsiz(sf_file file)
/*< return buffer size for efficient I/O >*/
{
   ...
}
```

Input parameters

`file` a pointer to the file structure whose buffer size is required (`sf_file`).

Output

`bufsiz` size of the buffer of the file structure.

5.1.11 sf_setform

Sets the form of the file, i.e. `SF_ASCII`, `SF_XDR`, `SF_NATIVE`.

Call

```
sf_setform (file, form);
```

Definition

```
void sf_setform (sf_file file, sf_dataform form)
/*< set file form >*/
{
   ...
}
```

Input parameters

`file` a pointer to the file structure whose form is to be set (`sf_file`).

`form` the type to be set. Must be of type `sf_datatype`, e.g. `SF_ASCII`.

5.1.12 sf_setformat

Sets the format of the file, e.g. `SF_INT`, `SF_FLOAT`, `SF_COMPLEX` etc. Format is the combination of file form and its type, e.g. `ASCII_INT`.

Call

```
sf_setformat (file, format);
```

Definition

```
void sf_setformat (sf_file file, const char* format)
/*< Set file format.
---
format has a form "form_type", i.e. native_float, ascii_int, etc.
>*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose format is to be set (`sf_file`).
`format`	the type to be set (`const char*`).

5.1.13 sf_getfilename

Returns a boolean value (true or false), depending on whether it was able to find the filename of an open file or not. The search is based on finding the file descriptor of an open file, if it is found the return value is `true`, otherwise `false`. Once the file name is found it is copied to the value pointed by the pointer `filename` which is given as input and is already defined in the `sf_input`.

Call

```
success = getfilename (fp, filename);
```

Definition

```
static bool getfilename (FILE* fp, char *filename)
/* Finds filename of an open file from the file descriptor.

Unix-specific and probably non-portable. */
{
   ...
}
```

Input parameters

`fp`	a pointer to the file structure whose file name is required (`FILE*`).
`filename`	pointer to the parameter on which the found file name is to be stored.

Output

`success`	a boolean value which is true if the filename is found otherwise false.

5.1.14 sf_gettmpdatapath

Returns the path of temporary data. It takes no input parameters. The places it looks for the temporary data path are listed in the function definition comment.

Call

```
path gettmpdatapath ();
```

Definition

```
static char* gettmpdatapath (void)
/* Finds temporary datapath.

Datapath rules:
1. check tmpdatapath= on the command line
```

```
2. check TMPDATAPATH environmental variable
3. check .tmpdatapath file in the current directory
4. check .tmpdatapath in the home directory
5. return NULL
*/
{
   ...
}
```

Output

`path` a pointer (of type `char`) to the value of the `tmpdatapath`.

5.1.15 sf_getdatapath

Returns the path of the data. It takes no input parameters. The places it looks for the temporary data path are listed in the function definition comment.

Call

```
path =  getdatapath();
```

Definition

```
static char* getdatapath (void)
/* Finds datapath.

Datapath rules:
1. check datapath= on the command line
2. check DATAPATH environmental variable
3. check .datapath file in the current directory
4. check .datapath in the home directory
5. use '.' (not a SEPlib behavior)
*/
{
   ...
}
```

Output

`path` a pointer (of type char) to the value of the `datapath`.

5.1.16 sf_readpathfile

Returns a boolean value (true or false), depending on whether it was able to find the data path in an open file or not. Once the `datapath` is found it is copied to the value pointed by the pointer `datapath` which is given as input and is already defined in the `sf_input`.

Call

```
success = readpathfile (filename, datapath);
```

Definition

```
static bool readpathfile (const char* filename, char* datapath)
/* find datapath from the datapath file */
{
   ...
}
```

Input parameters

`filename`	a pointer to the file name in which datapath is to be found (`const char`).
`datapath`	pointer to the parameter which is being looked for.

Output

`success`	a boolean value which is true if the filename is found otherwise false.

5.1.17 sf_fileclose

Closes the file and frees any allocated space, like the temporary file and buffer.

Call

```
sf_fileclose (file);
```

Definition

```
void sf_fileclose (sf_file file)
/*< close a file and free allocated space >*/
{
   ...
}
```

Input parameters

`file`	the file which is to be closed (`sf_file`).

5.1.18 sf_histint

Extracts an integer from the file. If the extraction is successful, it returns a `true`, otherwise a `false`. It uses `sf_simtab_getint`.

Call

```
success = sf_histint (file, key, par);
```

Definition

```
bool sf_histint (sf_file file, const char* key,/*@out@*/ int* par)
/*< read an int parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which an integer is to be extracted (`sf_file`).
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the integer variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.19 sf_histints

Extracts an array of integer values from the file. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_histints (file, key, par, n);
```

Definition

```
bool sf_histints (sf_file file, const char* key,/*@out@*/ int* par,size_t n)
/*< read an int array of size n parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which an integer array is to be extracted (`sf_file`).
`key`	the name of the integer array which has to be extracted (`const char*`).
`par`	pointer to the array of integer type value variable where the extracted value is to be copied.
`n`	size of the array to be extracted. Must be of `size_t`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.20 sf_histlargeint

Extracts a large integer from the file. If the extraction is successful, it returns a `true`, otherwise a `false`. It uses `sf_simtab_getlargeint`.

Call

```
success = sf_histlargeint ( file, key, par);
```

Definition

```
bool sf_histlargeint (sf_file file, const char* key,/*@out@*/ off_t* par)
/*< read a sf_largeint parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which a large integer is to be extracted (`sf_file`).
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the large integer variable where the extracted value is to be copied. Must be of type `off_t` which is defined in the header `<sys/types.h>`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.21 sf_histfloat

Extracts a float value from the file. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_histfloat (file, key, par);
```

Definition

```
bool sf_histfloat (sf_file file, const char* key,/*@out@*/ float* par)
/*< read a float parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which a floating point number is to be extracted (`sf_file`).
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the float type value variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.22 sf_histdouble

Extracts a double type value from the file. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_histdouble (file, key, par);
```

Definition

```
bool sf_histdouble (sf_file file, const char* key,/*@out@*/ double* par)
/*< read a float parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which a double type value is to be extracted (`sf_file`).
`table`	the table from which the vale has to be extracted. Must be of type `sf_simtab`.
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the double type value variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.23 sf_histfloats

Extracts an array of float values from the file. If the extraction is successful returns a `true`. It uses `sf_simtab_getfloats`.

Call

```
success = sf_histfloats(file, key, par, n);
```

Definition

```
bool sf_histfloats (sf_file file, const char* key,
          /*@out@*/ float* par,   size_t n)
/*< read a float array of size n parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which a float type array is to be extracted (`sf_file`).
`key`	the name of the float array which has to be extracted (`const char*`).
`par`	pointer to the array of float type value variable where the extracted value is to be copied.
`n`	size of the array to be extracted (`size_t`).

5.1.24 sf_histbool

Extracts a boolean value from the file. If the extraction is successful, it returns a `true`, otherwise a `false`.

Call

```
success = sf_histbool(file, key, par);
```

Definition

```
bool sf_histbool (sf_file file, const char* key,/*@out@*/ bool* par)
/*< read a bool parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which a bool type value is to be extracted (`sf_file`).
`key`	the name of the entry which has to be extracted (`const char*`).
`par`	pointer to the bool variable where the extracted value is to be copied.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.25 sf_histtbools

Extracts an array of bool values from the file. If the extraction is successful, it returns a `true`, otherwise a `false`.It uses `sf_simtab_getbools`.

Call

```
success = sf_histbools(file, key, par, n);
```

Definition

```
bool sf_histbools (sf_file file, const char* key,
                   /*@out@*/ bool* par, size_t n)
/*< read a bool array of size n parameter from file >*/
{
   ...
}
```

Input parameters

`file`	file from which an array of bool value is to be extracted (`sf_file`).
`key`	the name of the bool array which has to be extracted (`const char*`).
`par`	pointer to the array of bool type value variable where the extracted value is to be copied.
`n`	size of the array to be extracted. Must be of `size_t`.

Output

`success`	a boolean value. It is `true`, if the extraction was successful and `false` otherwise.

5.1.26 sf_histstring

Extracts a string pointed by the input key from the file. If the value is `NULL` it will return `NULL`, otherwise it will allocate a new block of memory of `char` type and copy the memory block from the table to the new block and return a pointer to the newly allocated block of memory. It uses `sf_simtab_getstring`.

Call

```
string = sf_histstring(file, key);
```

Definition

```
char* sf_histstring (sf_file file, const char* key)
/*< read a string parameter from file (returns NULL on failure) >*/
{
   ...
}
```

Input parameters

`file`	file from which a string is to be extracted (`sf_file`).
`key`	the name of the string which has to be extracted (`const char*`).

Output

`string`	a pointer to allocated block of memory containing a string of characters.

5.1.27 sf_fileflush

Outputs the parameters from source file to the output file. It sets the data format in the output file and prepares the file for writing binary data.

Call

```
sf_fileflush( file, src);
```

Definition

```
void sf_fileflush (sf_file file, sf_file src)
/*< outputs parameter to a file (initially from source src)
---
Prepares file for writing binary data >*/
{
...
}
```

Input parameters

`file`	pointer to the output file (`sf_file`).
`src`	a pointer to the input file structure (`sf_file`).

5.1.28 sf_putint

Enters an integer value in the file. It uses `sf_simtab_enter`.

Call

```
sf_putint (file, key, par);
```

Definition

```
void sf_putint (sf_file file, const char* key, int par)
/*< put an int parameter to a file >*/
{
   ...
}
```

Input parameters

`file`	the file in which the the key value is to be stored (`sf_file`).
`key`	pointer to the name of the key value to be input (`const char*`).
`par`	integer parameter which is to be written.

5.1.29 sf_putints

Enters an array of integer values in the file. It uses `sf_simtab_enter`.

Call

```
sf_putints (file, key, par, n);
```

Definition

```
void sf_putints (sf_file file, const char* key, const int* par, size_t n)
/*< put an int array of size n parameter to a file >*/
{
   ...
}
```

Input parameters

`file`	the file in which the the key value is to be stored (`sf_file`).
`key`	pointer to the name of the key value to be input (`const char*`).
`par`	pointer to integer parameter array which is to be written.
`n`	size of the array to be written (`size_t`).

5.1.30 sf_putlargeint

Enters a long integer value in the file. It uses `sf_simtab_enter`.

Call

```
sf_putlargeint (file, key, par);
```

Definition

```
void sf_putlargeint (sf_file file, const char* key, off_t par)
/*< put a sf_largeint parameter to a file >*/
{
   ...
}
```

Input parameters

`file`	the file in which the the key value is to be stored (`sf_file`).
`key`	pointer to the name of the key value to be input (`const char*`).
`par`	integer parameter which is to be written.

5.1.31 sf_putfloat

Enters a float value in the file. It uses `sf_simtab_enter`.

Call

```
sf_putfloat (file, key, par);
```

Definition

```
void sf_putfloat (sf_file file, const char* key,float par)
/*< put a float parameter to a file >*/
{
   ...
}
```

Input parameters

`file`	the file in which the the key value is to be stored (`sf_file`).
`key`	pointer to the name of the key value to be input (`const char*`).
`par`	floating point parameter which is to be written.

Definition

```
void sf_putfloat (sf_file file, const char* key,float par)
/*< put a float parameter to a file >*/
{
   ...
}
```

5.1.32 sf_putstring

Enters a string in to the file. It uses `sf_simtab_enter`.

Call

```
sf_putstring (file, key, par);
```

Definition

```
void sf_putstring (sf_file file, const char* key,const char* par)
/*< put a string parameter to a file >*/
{
   ...
}
```

Input parameters

`file`	the file in which the key value is to be stored (`sf_file`).
`key`	pointer to the name of the key value to be input (`const char*`).
`par`	pointer to the string parameter which is to be written.

5.1.33 sf_putline

Enters a string line in to the file.

Call

```
sf_putline (file, line);
```

Definition

```
void sf_putline (sf_file file, const char* line)
/*< put a string line to a file >*/
{
   ...
}
```

Input parameters

`file`	the file in which the string line is to be stored (`sf_file`).
`line`	pointer to the which is to be written.

5.1.34 sf_setaformat

Sets number format specifiers for ASCII output. This can be used in `sf_complexwrite`, for example.

Call

```
sf_setaformat (format, line);
```

Definition

```
void sf_setaformat (const char* format /* number format (.i.e "%5g") */,
                    int line /* numbers in line */ )
/*< Set format for ascii output >*/
{
   ...
}
```

Input parameters

`format`	a number format, e.g. %5g.
`line`	numbers in the ASCII line.

5.1.35 sf_complexwrite

Writes a complex array to the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_complexwrite (arr, size, file);
```

Definition

```
void sf_complexwrite (sf_complex* arr, size_t size, sf_file file)
/*< write a complex array arr[size] to file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array which is to be written (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.36 sf_complexread

Reads a complex array from the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_complexread (arr, size, file);
```

Definition

```
void sf_complexread (/*@out@*/ sf_complex* arr, size_t size, sf_file file)
/*< read a complex array arr[size] from file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array to which the array from the file is to be copied (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.37 sf_charwrite

Writes a character array to the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_charwrite (arr, size, file);
```

Definition

```
void sf_charwrite (char* arr, size_t size, sf_file file)
/*< write a char array arr[size] to file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array which is to be written (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.38 sf_uncharwrite

Writes a unsigned character array to the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_ucharwrite (arr, size, file);
```

Definition

```
void sf_ucharwrite (unsigned char* arr, size_t size, sf_file file)
/*< write an unsigned char array arr[size] to file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array which is to be written (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.39 sf_charread

Reads a character array from the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_charread (arr, size, file);
```

Definition

```
void sf_charread (/*@out@*/ char* arr, size_t size, sf_file file)
/*< read a char array arr[size] from file >*/
{
   ...
}
```

Input parameters

arr	a pointer to the array to which the array from the file is to be copied (sf_complex).
size	size of the array (size_t).
file	a file in which the array is to be written (sf_file).

5.1.40 sf_uncharread

Reads an unsigned character array from the file, according to the value of the form of the file, i.e. SF_ASCII, SF_XDR or SF_NATIVE.

Call

```
sf_ucharread (arr, size, file);
```

Definition

```
void sf_ucharread (/*@out@*/ unsigned char* arr, size_t size, sf_file file)
/*< read a uchar array arr[size] from file >*/
{
   ...
}
```

Input parameters

arr	a pointer to the array to which the array from the file is to be copied (sf_complex).
size	size of the array (size_t).
file	a file in which the array is to be written (sf_file).

5.1.41 sf_intwrite

Writes an integer array to the file, according to the value of the form of the file, i.e. SF_ASCII, SF_XDR or SF_NATIVE.

Call

```
sf_intwrite (arr, size, file);
```

Input parameters

arr	a pointer to the array which is to be written (sf_complex).
size	size of the array (size_t).
file	a file in which the array is to be written (sf_file).

5.1.42 sf_intread

Reads an integer array from the file, according to the value of the form of the file, i.e. SF_ASCII, SF_XDR or SF_NATIVE.

Call

```
sf_intread (arr, size, file);
```

Definition

```
void sf_intread (/*@out@*/ int* arr, size_t size, sf_file file)
/*< read an int array arr[size] from file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array to which the array from the file is to be copied (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.43 sf_shortread

Reads an short array from the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_shortread (arr, size, file);
```

Definition

```
void sf_shortread (/*@out@*/ short* arr, size_t size, sf_file file)
/*< read a short array arr[size] from file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array to which the array from the file is to be copied (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.44 sf_shortwrite

Writes an short array to the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_shortwrite (arr, size, file);
```

Definition

```
void sf_shortwrite (short* arr, size_t size, sf_file file)
/*< write a short array arr[size] to file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array which is to be written (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.45 sf_floatwrite

Writes an float array to the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_floatwrite (arr, size, file);
```

Definition

```
void sf_floatwrite (float* arr, size_t size, sf_file file)
/*< write a float array arr[size] to file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array which is to be written (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.46 sf_floatread

Reads a float array from the file, according to the value of the form of the file, i.e. `SF_ASCII`, `SF_XDR` or `SF_NATIVE`.

Call

```
sf_floatread (arr, size, file);
```

Definition

```
void sf_floatread (/*@out@*/ float* arr, size_t size, sf_file file)
/*< read a float array arr[size] from file >*/
{
   ...
}
```

Input parameters

`arr`	a pointer to the array to which the array from the file is to be copied (`sf_complex`).
`size`	size of the array (`size_t`).
`file`	a file in which the array is to be written (`sf_file`).

5.1.47 sf_bytes

Returns the size of the file in bytes.

Call

```
size = sf_bytes (file);
```

Definition

```
off_t sf_bytes (sf_file file)
/*< Count the file data size (in bytes) >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose size is required (`sf_file`).

Output

`size`	the size of the file structure in bytes.

5.1.48 sf_tell

Returns the current value of the position indicator of the file.

Call

```
val = sf_tell (file);
```

Definition

```
off_t sf_tell (sf_file file)
/*< Find position in file >*/
{
   ...
}
```

Input parameters

`file` a pointer to the file structure the value of whose position indicator required (`sf_file`).

Output

Current value of the position indicator of the file. It is of type `off_t`.

5.1.49 sf_tempfile

Creates a temporary file with a unique file name.

Call

```
tmp = sf_tempfile(dataname, mode);
```

Definition

```
FILE *sf_tempfile(char** dataname, const char* mode)
/*< Create a temporary file with a unique name >*/
{
   ...
}
```

Input parameters

`dataname`	a pointer to the value of the name of the temporary file (`char**`).
`mode`	mode of the file to be created, e.g. `w+`.

Output

`tmp` a pointer to the temporary file.

5.1.50 sf_seek

Call

```
sf_seek (file, offset, whence);
```

Definition

```
void sf_seek (sf_file file, off_t offset, int whence)
/*< Seek to a position in file. Follows fseek convention. >*/
{
   ...
}
```

5.1.51 sf_unpipe

Redirects a pipe input to a directly accessible file.

Call

```
sf_unpipe (file, size);
```

Definition

```
void sf_unpipe (sf_file file, off_t size)
/*< Redirect a pipe input to a direct access file >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure which is to be unpiped (`sf_file`).

5.1.52 sf_close

Removes temporary files.

Call

```
sf_close();
```

Definition

```
void sf_close(void)
/*< Remove temporary files >*/
{
   ...
}
```

5.2 Additional operations with RSF files (files.c)

5.2.1 sf_filedims

Returns the dimensions of the file.

Call

```
dim = sf_filedims (file, n);
```

Definition

```
int sf_filedims (sf_file file, /*@out@*/ int *n)
/*< Find file dimensions.
---
Outputs the number of dimensions dim and a dimension array n[dim] >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose dimensions are required (`sf_file`).
`n`	an array where the dimensions will be stored (`int`).

Output

`dim`	number of dimensions in the file (`int`).
`n[dim]`	the array of dimensions (`int`).

5.2.2 sf_largefiledims

Returns the dimensions of the file. It is exactly like `sf_filedims` but `n` in the input is the offset in bytes in the input file (type `off_t`) rather than just an integer.

Call

```
dim = sf_largefiledims (file, n);
```

Definition

```
int sf_largefiledims (sf_file file, /*@out@*/ off_t *n)
/*< Find file dimensions.
---
Outputs the number of dimensions dim and a dimension array n[dim] >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose dimensions are required (`sf_file`).
`n`	an array where the dimensions will be stored (`off_t`).

Output

`dim`	number of dimensions in the file (`off_t`).
`n[dim]`	the array of dimensions (`off_t`).

5.2.3 sf_memsize

Returns the memory size defined in the environment variable `RSFMEMSIZE`. If there is an invalid value the function will print an error message an assign a default value of 100 Mbytes.

Call

```
memsize = sf_memsize();
```

Definition

```
int sf_memsize()
/*< Returns memory size by:
  1. checking RSFMEMSIZE environmental variable
  2. using hard-coded "def" constant
  >*/
{

   ...
    return memsize;
}
```

5.2.4 sf_filesize

Returns the size of the file, that is, the product of the dimensions. It uses the function `sf_leftsize`.

Call

```
size = sf_filesize (file);
```

Definition

```
off_t sf_filesize (sf_file file)
/*< Find file size (product of all dimensions) >*/
{
   ...
}
```

Input parameters

`file`	a pointer to the file structure whose size is required (`sf_file`).

Output

`size`	product of dimensions in the file (`off_t`).

5.2.5 sf_leftsize

Returns the size of the file, that is, the product of the dimensions but only for the dimensions greater than the input integer dim. It uses the function `sf_leftsize`.

Call

```
size = sf_leftsize (file, dim);
```

Definition

```
off_t sf_leftsize (sf_file file, int dim)
/*< Find file size for dimensions greater than dim >*/
{
   ...
}
```

Input parameters

`file`	the file whose size is required (`sf_file`).
`dim`	a pointer to the file structure whose size is required (`sf_file`).

Output

`size`	product of dimensions greater than dim in the file (`off_t`).

5.2.6 sf_cp

Copies the input file `in` to the output file `out`.

Call

```
sf_cp (in, out);
```

Definition

```
void sf_cp(sf_file in, sf_file out)
/*< Copy file in to file out >*/
{
   ...
}
```

Input parameters

`in`	the file which is to be copied (`sf_file`).
`out`	the file to which `in` file is to be copied (`sf_file`).

5.2.7 sf_rm

Removes the RSF file. There are options to force removal (files are deleted even if protected), to inquire before removing a file and whether or to require verbose output.

Call

```
sf_rm (filename, force, verb, inquire);
```

Definition

```
void sf_rm(const char* filename, bool force, bool verb, bool inquire)
/*< Remove an RSF file.
---
force, verb, and inquire flags should behave similar to the corresponding flags
in the Unix "rm" command. >*/
{
   ...
}
```

Input parameters

`filename`	name of the file which is to be removed (`sf_file`).
`force`	remove forcefully or not (`sf_file`).
`inquire`	ask before removing or not (`sf_file`).

5.2.8 sf_shiftdim

Shifts the grid by one dimension after the axis defined in the input parameters (axis).

Call

```
n3 = sf_shiftdim(in, out, axis);
```

Definition

```
off_t sf_shiftdim(sf_file in, sf_file out, int axis)
/*< shift grid after axis by one dimension forward >*/
{
   ...
}
```

Input parameters

`in`	a pointer to the input file structure (`sf_file`).
`out`	a pointer to the output file structure (`sf_file`).
`axis`	the axis after which the grid is to be shifted (`sf_file`).

Output

`n3`	the file size (product of dimensions) after the shift (`off_t`).

5.2.9 sf_unshiftdim

Shifts the grid backward by one dimension after the axis defined in the input parameters (axis).

Call

```
n3 = sf_unshiftdim (in, out, axis);
```

Definition

```
off_t sf_unshiftdim(sf_file in, sf_file out, int axis)
/*< shift grid after axis by one dimension backward >*/
{
   ...
}
```

Input parameters

`in`	a pointer to the input file structure (`sf_file`).
`out`	a pointer to the output file structure (`sf_file`).
`axis`	the axis after which the grid is to be shifted (`sf_file`).

Output

`n3`	the file size (product of dimensions) after the backward shift (`off_t`).

5.2.10 sf_endian

Returns true if the machine is little endian.

Definition

```
little_endian = sf_endian ();
```

Definition

```
bool sf_endian (void)
/*< Endianness test, returns true for little-endian machines >*/
{
   ...
}
```

Output

`little_endian`	a boolean parameter which is true if the machine is little endian.

5.3 Complex number operations (komplex.c)

5.3.1 creal

Returns the real part of the complex number.

Call

```
r = sf_creal(c);
```

Definition

```
double sf_creal(sf_double_complex c)
/*< real part >*/
{
   ...
}
```

Input parameters

`c` a complex number. Must be of type `sf_double_complex`.

Output

`r` real part of the complex number. It is of type `double`.

5.3.2 cimag

Returns the imaginary part of the complex number.

Call

```
im = sf_cimag(c);
```

Definition

```
double sf_cimag(sf_double_complex c)
/*< imaginary part >*/
{
   ...
}
```

Input parameters

`c` a complex number. Must be of type `sf_double_complex`.

Output

`im` imaginary part of the complex number (`double`).

5.3.3 dcneg

Returns the negative complex number.

Call

```
n = sf_dcneg(a);
```

Definition

```
sf_double_complex sf_dcneg(sf_double_complex a)
/*< unary minus >*/
{
   ...
}
```

Input parameters

`a`	a complex number. Must be of type `sf_double_complex`.

Output

`n`	negative of the complex number. It is of type `sf_double_complex`.

5.3.4 dcadd

Adds two complex numbers.

Call

```
c = sf_dcadd(a, b);
```

Definition

```
sf_double_complex sf_dcadd(sf_double_complex a, sf_double_complex b)
/*< complex addition >*/
{
   ...
}
```

Input parameters

`a`	a complex number. Must be of type `sf_double_complex`.
`b`	a complex number. Must be of type `sf_double_complex`.

Output

`c`	$a + b$. It is of type `sf_double_complex`.

5.3.5 dcsub

Subtracts two complex numbers.

Call

```
c = sf_dcsub(a, b);
```

Definition

```
sf_double_complex sf_dcsub(sf_double_complex a, sf_double_complex b)
/*< complex subtraction >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `sf_double_complex`.

`b` a complex number. Must be of type `sf_double_complex`.

Output

`c` $a - b$. It is of type `sf_double_complex`.

5.3.6 dcmul

Multiplies two complex number.

Call

```
c = sf_dcmul(a, b);
```

Definition

```
sf_double_complex sf_dcmul(sf_double_complex a, sf_double_complex b)
/*< complex multiplication >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `sf_double_complex`.

`b` a complex number. Must be of type `sf_double_complex`.

Output

`c` the product ab. It is of type `sf_double_complex`.

5.3.7 dccmul

Multiplies two complex number. Its output type and one of the input parameters is of type `kiss_fft_cpx`.

Call

```
c = sf_dccmul(a, b);
```

Definition

```
kiss_fft_cpx sf_dccmul(sf_double_complex a, kiss_fft_cpx b)
/*< complex multiplication >*/
{
   ...
}
```

Input parameters

`a`	a complex number. Must be of type `sf_double_complex`.
`b`	a complex number. Must be of type `kiss_fft_cpx`.

Output

`c`	the product *ab*. It is of type `sf_double_complex`.

5.3.8 dcdmul

Multiplies two complex number. One of the input parameters is `kiss_fft_cpx`. This means that it should only be used if `complex.h` header is not used.

Call

```
c = sf_dcdmul(a, b);
```

Definition

```
sf_double_complex sf_dcdmul(sf_double_complex a, kiss_fft_cpx b)
/*< complex multiplication >*/
{
   ...
}
```

Input parameters

`a`	a complex number. Must be of type `sf_double_complex`.
`b`	a complex number. Must be of type `kiss_fft_cpx`.

Output

`c`	product *ab* of the two complex numbers *a* and *b*. It is of type `sf_double_complex`.

5.3.9 dcrmul

Multiplies a complex number with a real number of type `double`.

Call

```
c = sf_dcrmul(a, b);
```

Definition

```
sf_double_complex sf_dcrmul(sf_double_complex a, double b)
/*< complex by real multiplication >*/
{
   ...
}
```

Input parameters

`a`	a complex number. Must be of type `sf_double_complex`.
`b`	a real number (`double`).

Output

`c`	product of the complex number *a* and the real number *b*. It is of type `sf_double_complex`.

5.3.10 dcdiv

Divides two complex numbers.

Call

```
c = sf_dcdiv(a, b);
```

Definition

```
sf_double_complex sf_dcdiv(sf_double_complex a, sf_double_complex b)
/*< complex division >*/
{
   ...
}
```

Input parameters

`a`	a complex number. Must be of type `sf_double_complex`.
`b`	a complex number. Must be of type `sf_double_complex`.

Output

`c`	$\frac{a}{b}$. It is of type `sf_double_complex`.

5.3.11 cabs

Returns the absolute value (magnitude) of a complex number. It uses the `hypot` function from the C library.

Call

```
a = sf_cabs(z);
```

Definition

```
double sf_cabs(sf_double_complex z)
/*< replacement for cabsf >*/
{
   ...
}
```

Input parameters

`z`	a complex number. Must be of type `sf_double_complex`.

Output

`hypot(z.r,z.i)`	absolute value of the complex number.

5.3.12 cabs

Returns the argument of a complex number. It uses the `atan2` function from the C library.

Call

```
u = sf_carg(z);
```

Definition

```
double sf_carg(sf_double_complex z)
/*< replacement for cargf >*/
{
   ...
}
```

Input parameters

`z`	a complex number. Must be of type `sf_double_complex`.

Output

`atan2(z.r,z.i)`	argument of the complex number.

5.3.13 crealf

Returns the real part of the complex number.

Call

```
r = sf_crealf(c);
```

Definition

```
float sf_crealf(kiss_fft_cpx c)
/*< real part >*/
{
   ...
}
```

Input parameters

`c` a complex number. Must be of type `kiss_fft_cpx`.

Output

`r` real part of the complex number. It is of type `float`.

5.3.14 cimagf

Returns the imaginary part of the complex number.

Call

```
im = sf_cimagf(c);
```

Definition

```
float sf_cimagf(kiss_fft_cpx c)
/*< imaginary part >*/
{
   ...
}
```

Input parameters

`c` a complex number. Must be of type `kiss_fft_cpx`.

Output

`im` imaginary part of the complex number. It is of type `float`.

5.3.15 cprint

Prints the complex number on the screen. This is done using the `sf_warning`.

Call

```
cprint(c);
```

Definition

```
void cprint (sf_complex c)
/*< print a complex number (for debugging purposes) >*/
{
   ...
}
```

Input parameters

`c` a complex number (`sf_complex`).

5.3.16 cadd

Adds two complex numbers. The output is of type `kiss_fft_cpx`.

Call

```
c = sf_cadd(a, b);
```

Definition

```
kiss_fft_cpx sf_cadd(kiss_fft_cpx a, kiss_fft_cpx b)
/*< complex addition >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `kiss_fft_cpx`.

`b` a complex number. Must be of type `kiss_fft_cpx`.

Output

`c` the sum $a + b$ of the two complex numbers a, b. It is of type `kiss_fft_cpx`.

5.3.17 csub

Subtracts two complex numbers. The output is of type `kiss_fft_cpx`.

Call

```
c = sf_csub(a, b);
```

Definition

```
kiss_fft_cpx sf_csub(kiss_fft_cpx a, kiss_fft_cpx b)
/*< complex subtraction >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `kiss_fft_cpx`.

`b` a complex number. Must be of type `kiss_fft_cpx`.

Output

`c` difference of the two complex numbers a, b. It is of type `kiss_fft_cpx`.

5.3.18 csqrtf

Returns the square root of a complex number. The output is of type `kiss_fft_cpx`.

Call

```
c = sf_csqrtf (c);
```

Definition

```
kiss_fft_cpx sf_csqrtf (kiss_fft_cpx c)
/*< complex square root >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `kiss_fft_cpx`.

`b` a complex number. Must be of type `kiss_fft_cpx`.

Output

`c` square root of the complex number. It is of type `kiss_fft_cpx`.

5.3.19 cdiv

Divides two complex numbers. The output is of type `kiss_fft_cpx`.

Call

```
c = sf_cdiv(a, b);
```

Definition

```
kiss_fft_cpx sf_cdiv(kiss_fft_cpx a, kiss_fft_cpx b)
/*< complex division >*/
{
   ...
}
```

Input parameters

a a complex number. Must be of type sf_double_complex.

b a complex number. Must be of type sf_double_complex.

Output

c $\frac{a}{b}$. It is of type kiss_fft_cpx.

5.3.20 cmul

Multiplies two complex numbers. The output is of type kiss_fft_cpx.

Call

```
c = sf_cmul(a, b);
```

Definition

```
kiss_fft_cpx sf_cmul(kiss_fft_cpx a, kiss_fft_cpx b)
/*< complex multiplication >*/
{
   ...
}
```

Input parameters

a a complex number. Must be of type sf_double_complex.

b a complex number. Must be of type sf_double_complex.

Output

c product of the two complex numbers a and b. It is of type kiss_fft_cpx.

5.3.21 crmul

Multiplies a complex number with a real number. The output is of type kiss_fft_cpx.

Call

```
c = sf_crmul(a, b);
```

Definition

```
kiss_fft_cpx sf_crmul(kiss_fft_cpx a, float b)
/*< complex by real multiplication >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `sf_double_complex`.

`b` a real number (`float`).

Output

`c` the product ab of a complex number a and real number b. It is of type `kiss_fft_cpx`.

5.3.22 cneg

Returns negative of a complex number. The output is of type `kiss_fft_cpx`.

Call

```
b = sf_cneg(a);
```

Definition

```
kiss_fft_cpx sf_cneg(kiss_fft_cpx a)
/*< unary minus >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `sf_double_complex`.

Output

`a` negative of a complex number. It is of type `kiss_fft_cpx`.

5.3.23 conjf

Returns complex conjugate of a complex number. The output is of type `kiss_fft_cpx`.

Call

```
z1 = sf_conjf(z);
```

Definition

```
kiss_fft_cpx sf_conjf(kiss_fft_cpx z)
/*< complex conjugate >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `sf_double_complex`.

Output

`z1` complex conjugate of the complex number. It is of type `kiss_fft_cpx`.

5.3.24 cabsf

Returns the magnitude of a complex number. It uses a function `hypotf` from `c99.h`, which calls the `hypot` function from `math.h` in the C library.

Call

```
w = sf_cabsf(kiss_fft_cpx z);
```

Definition

```
float sf_cabsf(kiss_fft_cpx z)
/*< replacement for cabsf >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` magnitude of a complex number. It is of type `kiss_fft_cpx`.

5.3.25 cargf

Returns the argument of a complex number. It uses a function `atan2f` from `c99.h`, which calls the `atan2` function from `math.h` in the C library.

Call

```
u = sf_cargf(z);
```

Definition

```
float sf_cargf(kiss_fft_cpx z)
/*< replacement for cargf >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`u` argument of a complex number. It is of type `kiss_fft_cpx`.

5.3.26 ctanhf

Returns hyperbolic tangent of a complex number. It uses a function like `coshf` and `sinhf` from `c99.h`, which call `cosh` and `sinh` functions from `math.h` in the C library.

Call

```
th = sf_ctanhf(z);
```

Definition

```
kiss_fft_cpx sf_ctanhf(kiss_fft_cpx z)
/*< complex hyperbolic tangent >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`th` hyperbolic tangent of a complex number. It is of type `kiss_fft_cpx`.

5.3.27 ccosf

Returns cosine of a complex number. It uses the functions like `coshf` and `sinhf` from `c99.h`, which call `cosh` and `sinh` functions from `math.h` in the C library.

Call

```
w =  sf_ccosf (z);
```

Definition

```
kiss_fft_cpx sf_ccosf(kiss_fft_cpx z)
/*< complex cosine >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` cosine of a complex number. It is of type `kiss_fft_cpx`.

5.3.28 ccoshf

Returns hyperbolic cosine of a complex number. It uses the functions like `coshf` and `sinhf` from `c99.h`, which call `cosh` and `sinh` functions from `math.h` in the C library.

Call

```
w = sf_ccoshf(z);
```

Definition

```
kiss_fft_cpx sf_ccoshf(kiss_fft_cpx z)
/*< complex hyperbolic cosine >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` huperbolic cosine of a complex number. It is of type `kiss_fft_cpx`.

5.3.29 ccosf

Returns sine of a complex number. It uses the functions like `coshf` and `sinhf` from `c99.h`, which call `cosh` and `sinh` functions from `math.h` in the C library.

Call

```
w = sf_csinf(z);
```

Definition

```
kiss_fft_cpx sf_csinf(kiss_fft_cpx z)
/*< complex sine >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` sine of a complex number. It is of type `kiss_fft_cpx`.

5.3.30 csinhf

Returns hyperbolic cosine of a complex number. It uses the functions like `coshf` and `sinhf` from `c99.h`, which call `cosh` and `sinh` functions from `math.h` in the C library.

Call

```
w = sf_csinhf(z);
```

Definition

```
kiss_fft_cpx sf_csinhf(kiss_fft_cpx z)
/*< complex hyperbolic sine >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` huperbolic cosine of a complex number. It is of type `kiss_fft_cpx`.

5.3.31 clogf

Returns natural logarithm of a complex number. It uses the functions like `logf` and `hypotf` from `c99.h`, which call `log` and `hypot` functions from `math.h` in the C library.

Call

```
w = sf_clogf(z);
```

Definition

```
kiss_fft_cpx sf_clogf(kiss_fft_cpx z)
/*< complex natural logarithm >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` natural logarithm of a complex number. It is of type `kiss_fft_cpx`.

5.3.32 cexpf

Returns exponential of a complex number. It uses the functions like `expf` and `cosf` from `c99.h`, which call `exp` and `cos` functions from `math.h` in the C library.

Call

```
w = sf_cexpf(z);
```

Definition

```
kiss_fft_cpx sf_cexpf(kiss_fft_cpx z)
/*< complex exponential >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`z` exponential of a complex number. It is of type `kiss_fft_cpx`.

5.3.33 ctanf

Returns tangent of a complex number. It uses the functions like `sinf` and `cosf` from `c99.h`, which call `sin` and `cos` functions from `math.h` in the C library.

Call

```
w = sf_ctanf(z);
```

Definition

```
kiss_fft_cpx sf_ctanf(kiss_fft_cpx z)
/*< complex tangent >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` tangent of a complex number. It is of type `kiss_fft_cpx`.

5.3.34 casinf

Returns hyperbolic arcsine of a complex number. It uses the function `asinf` from `c99.h`, which calls the `asin` function from `math.h` in the C library.

Call

```
w = sf_casinf(z);
```

Definition

```
kiss_fft_cpx sf_casinf(kiss_fft_cpx z)
/*< complex hyperbolic arcsine >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` arcsine of a complex number. It is of type `kiss_fft_cpx`.

5.3.35 cacosf

Returns hyperbolic arccosine of a complex number. It uses `sf_cacosf`.

Call

```
w = sf_cacosf(z);
```

Definition

```
kiss_fft_cpx sf_cacosf(kiss_fft_cpx z)
/*< complex hyperbolic arccosine >*/
{
   ...
}
```

Input parameters

`z` a complex number. Must be of type `kiss_fft_cpx`.

Output

`w` hyperbolic arccosine of a complex number. It is of type `kiss_fft_cpx`.

5.3.36 catanf

Returns arctangent of a complex number. It uses `sf_clogf` and `sf_cdiv`.

Call

```
w = sf_catanf(z);
```

Definition

```
kiss_fft_cpx sf_catanf(kiss_fft_cpx z)
/*< complex arctangent >*/
{
   ...
}
```

Input parameters

z	a complex number. Must be of type `kiss_fft_cpx`.

Output

w	arctangent of a complex number. It is of type `kiss_fft_cpx`.

5.3.37 catanhf

Returns hyperbolic arctangent of a complex number. It uses `sf_catanf`.

Call

```
w =_cpx sf_catanhf(z);
```

Definition

```
kiss_fft_cpx sf_catanhf(kiss_fft_cpx z)
/*< complex hyperbolic arctangent >*/
{
   ...
}
```

Input parameters

z	a complex number. Must be of type `kiss_fft_cpx`.

Output

z	hyperbolic arctangent of a complex number. It is of type `kiss_fft_cpx`.

5.3.38 casinhf

Returns hyperbolic arcsine of a complex number. It uses `sf_casinf`.

Call

```
w = sf_casinhf(z);
```

Definition

```
kiss_fft_cpx sf_casinhf(kiss_fft_cpx z)
/*< complex hyperbolic sine >*/
{
   ...
}
```

Input parameters

`z`	a complex number. Must be of type `kiss_fft_cpx`.

Output

`z`	hyperbolic arcsine of a complex number. It is of type `kiss_fft_cpx`.

5.3.39 cacoshf

Returns hyperbolic arccosine of a complex number. It uses `sf_casinf`.

Call

```
w = sf_cacoshf(z);
```

Definition

```
kiss_fft_cpx sf_cacoshf(kiss_fft_cpx z)
/*< complex hyperbolic cosine >*/
{
   ...
}
```

Input parameters

`z`	a complex number. Must be of type `kiss_fft_cpx`.

Output

`w`	hyperbolic arccosine of a complex number. It is of type `kiss_fft_cpx`.

5.3.40 cpowf

Returns the complex base a raised to complex power b.

Call

```
c = sf_cpowf(a, b);
```

Definition

```
kiss_fft_cpx sf_cpowf(kiss_fft_cpx a, kiss_fft_cpx b)
/*< complex power >*/
{
   ...
}
```

Input parameters

`a` a complex number. Must be of type `kiss_fft_cpx`.

`b` a complex number. Must be of type `kiss_fft_cpx`.

Output

`c` a^b. It is a complex number of type `kiss_fft_cpx`.

Chapter 6

Error handling

6.1 Handling warning and error messages (error.c)

6.1.1 sf_error

Outputs an error message to `stderr`, which is usually the screen. It uses `sf_getprog` to get the name of the program which is causing the error and print it on the screen. Uses `vfprintf`, which can take a variable number of arguments initialized by `va_list`. This gives the user flexibility in choosing the number of arguments.

If there is a '`:`' at the end of format, information about the system errors is printed, this is done by using `strerror` to interpret the last error number `errno` in the system. Also, if there is a '`;`' at the end of a format the command prompt will not go to the next line.

Call

```
sf_error (format, ...);
```

Definition

```
void sf_error( const char *format, ... )
/*< Outputs an error message to stderr and terminates the program.
    ---
    Format and variable arguments follow printf convention. Additionally, a ':' at
    the end of format adds system information for system errors. >*/
{
   ...
}
```

Input parameters

`format`	a string of `type const char*` containing the format specifiers for the arguments to be input from the next commands.
`...`	variable number of arguments, which are to replace the format specifiers in the format.

Output

An error message output to `sterr` (usually printed on screen).

6.1.2 sf_warning

Outputs a warning message to `stderr` which is usually the screen. It uses `sf_getprog` to get the name of the program which is causing the error and print it on the screen. Uses `vfprintf`, which can take a variable number of arguments initialized by `va_list`. This gives the user flexibility in choosing the number of arguments. If there is a '`:`' at the end of format, information about the system errors is printed, this is done by using `strerror` to interpret the last error number `errno` in the system. Also, if there is a '`;`' at the end of a format the command prompt will not go to the next line.

Call

```
sf_warning (format, ... );
```

Definition

```
void sf_warning( const char *format, ... )
/*< Outputs a warning message to stderr.
---
Format and variable arguments follow printf convention. Additionally, a ':' at
the end of format adds system information for system errors. >*/
{
   ...
}
```

Input parameters

`format`	a string of `type const char*` containing the format specifiers for the arguments to be input from the next commands.
`...`	variable number of arguments, which are to replace the format specifiers in the format.

Output

A warning message output to `sterr` (usually printed on screen).

Chapter 7

Linear operators

7.1 Introduction

This section contains a bunch of programs that implement operators. Therefore a short introduction on operators is in order.

7.1.1 Definition of operators

Mathematically speaking an operator is a function of a function, i.e. a rule (or mapping) according to which a function f is transformed into another function g. We use the notation $g = R[f]$ or simply $g = Rf$, where R denotes the operator. Examples of operators are the derivative, the integral, convolution (with a specific function), multiplication by a scalar and others. Note that in general the domains of f and g are not necessarily the same. For example, in the case of the derivative, the domain of $g = Rf$ is the subset of the domain of f, in which f is smooth. In particular if $f = |x|$, $x \in [-1, 1]$, then the domain of g is $(-1, 0) \cup (0, 1)$.

An important class of operators are the **linear operators**. An operator L is linear if for any two functions f_1, f_2 and any two scalars a_1, a_2, $L[a_1 f_1 + a_1 f_2] = a_1 L f_1 + a_2 L f_2$. The derivative, integral, convolution and multiplication by scalar are all linear operators.

In the discrete world, operators act on vectors and linear operators are in fact matrices, with which the vectors are multiplied. (Multiplication by a matrix is a linear operation, since $\mathbf{M}(a_1\mathbf{x}_1 + a_2\mathbf{x}_2) = a_1\mathbf{M}\mathbf{x}_1 + a_2\mathbf{M}\mathbf{x}_2$). In fact many of the calculations performed routinely in science and engineering are essentially matrix multiplications in disguise. For example assume a vector $\mathbf{x} = [x_1\ x_2\ \dots\ x_n]^T$ with length n (superscript ^{T} denotes transpose). Padding this vector with m zeros, produces another vector $\mathbf{y}$ with

$$\mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix},$$

where $\mathbf{0}$ is the zero vector of length m. One can readily verify that zero padding is a linear operation with operator matrix $\mathbf{L} = \begin{bmatrix}\mathbf{I}\\ \mathbf{O}\end{bmatrix}$, where $\mathbf{I}$ is the $n \times n$ identity matrix and $\mathbf{O}$ is the $m \times n$ zero matrix, since

$$\mathbf{y} = \mathbf{L}\mathbf{x} = \begin{bmatrix}\mathbf{I}\\ \mathbf{O}\end{bmatrix}\mathbf{x} = \begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix}.$$

Note that as in the case of functions, the domains of $\mathbf{x}$ and $\mathbf{y}$ are different: $\mathbf{x} \in \mathbb{R}^n$ (or more generally $\mathbf{x} \in \mathbb{C}^n$), while $\mathbf{y} \in \mathbb{R}^{n+m}$ (or $\mathbb{C}^{n+m}$).

Similarly, one can define convolution of $\mathbf{x}$ with $\mathbf{a} = [a_1\ a_2\ \ldots\ a_m]^T$ as the multiplication of $\mathbf{x}$ with

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & 0 & \cdots & 0 & 0 \\ a_3 & a_2 & a_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & a_m & a_{m-1} \\ 0 & 0 & 0 & \cdots & 0 & a_m \end{bmatrix}.$$

and many other operations as matrix multiplications. Other operators are the identity operator is the identity matrix $\mathbf{I}$ and is implemented by `copy.c` and `ccopy.c` and the null operator (or zero matrix $\mathbf{O}$), which is implemented by `adjnull.c`. For the rest of this introduction, the boldface notation will imply specifically discrete operators, while the normal fonts will imply operators on either continuous or discrete mathematical entities.

7.1.2 Products of operators

The result of an operation on a function is another function, therefore we can naturally apply an operator on another operator. In other words, if L_1, L_2 are two operators, then we can define L_1L_2 as $L_1L_2[x] = L_1[L_2[x]]$, provided that $L_1[L_2[x]]$ makes sense mathematically. This is called the composition of the operators L_1 and L_2. Because in the discrete case the composition of operators is in fact the multiplication L_1L_2 of the two matrices L_1, L_2 the operator composition is usually referred to as operator product and denoted by L_1L_2 is used. The composition of operators can be naturally extended to any finite product $L_1 \cdots L_{n-1}L_n$. The product of up to 3 operators is implemented in `chain.c`.

7.1.3 Adjoint operators

A very important notion in data processing is the **adjoint operator**[1] L^* of an operator L. In the discrete world, the adjoint operator of L is its (conjugate) transpose, i.e. $L^* = L^H$. From this definition of the adjoint it is evident that *the adjoint of the adjoint is the original operator* (since $(L^*)^* = (L^H)^H = L$). Consider a vector $\mathbf{y} = [y_1\, y_2, \ldots, y_{n+m}]^T$ and the adjoint of the

[1]The adjoint operator should not be confused with the (classical) adjoint or adjucate or adjunct matrix of a square matrix. The adjugate matrix of an invertible matrix is the inverse multiplied by its determinant.

zero-padding operator, $\mathbf{L}^* = \mathbf{L}^H = \begin{bmatrix}\mathbf{I}\\ \mathbf{O}\end{bmatrix}^T = \begin{bmatrix}\mathbf{I} & \mathbf{O}^T\end{bmatrix}$. Then

$$\mathbf{L}^*\mathbf{y} = \begin{bmatrix}\mathbf{I} & \mathbf{O}^T\end{bmatrix}\mathbf{y} = \begin{bmatrix}\mathbf{I} & \mathbf{O}^T\end{bmatrix}\begin{bmatrix}y_1\\ y_2\\ \vdots\\ y_n\\ y_{n+1}\\ \vdots\\ y_{n+m}\end{bmatrix} = \begin{bmatrix}y_1\\ y_2\\ \vdots\\ y_n\end{bmatrix}.$$

We conclude that the adjoint of data zero-padding is data truncation. It is also easy (but tedious) to verify that the adjoint operation of the convolution between $\mathbf{a}$ and $\mathbf{x}$ is the crosscorrelation of $\mathbf{a}$ with $\mathbf{y}^H$. One may also notice that for the specific zero-padding operator, $\mathbf{L}^*\mathbf{L}\mathbf{x} = \mathbf{x}$, i.e. in this case the adjoint neutralizes the effect of the operator. It is tempting to say that the adjoint operation is the inverse operation, however this is not the case: it is not always the case that $L^*L = LL^*$. In fact such an equality is meaningless mathematically if $\mathbf{L}$ is not a square matrix (if $\mathbf{L}$ is a $n \times m$ matrix, then $\mathbf{L}\mathbf{L}^*$ is $n \times n$, while $\mathbf{L}^*\mathbf{L}$ is $m \times m$). L^* is not even the left inverse of L: notice that in the case of the zero padding operator, $(\mathbf{L}^*)^*\mathbf{L}^*\mathbf{y} = \mathbf{L}\mathbf{L}^*\mathbf{y} = \tilde{\mathbf{y}} \neq \mathbf{y}$ (the last m elements of $\tilde{\mathbf{y}}$ are zero). However it is often the case that the adjoint is an adequate. It is the case though quite often that the adjoint is adequate approximation to the inverse (sometimes within a scaling factor) and it is also quite probable that the adjoint will do a better job than the inverse in inverse problems. This is because the adjoint operator tolerates data imperfections, which the inverse does not.

From the definition of the adjoint operation as the left multiplication the complex conjugate matrix, it follows that *the adjoint of the product of two linear operators equals the product of the adjoints in reverse order*, i.e. $(L_1L_2)^* = L_2^*L_1^*$. This is naturally extended to the product of any finite product of operators, i.e. $(L_1L_2\cdots L_n)^* = L_n^*L_{n-1}^*\cdots L_1^*$. The adjoint of the product is also implemented in `chain.c`.

7.1.4 The dot-product test

The dot-product test is a valuable checkpoint, which can tell us whether the implementation of the adjoint operator is wrong (however it cannot guarantee that it is indeed correct). The concept is the following: Assuming that we have coded an operator L and its adjoint L^*. Then for any two vectors or functions a and b,

$$\langle a, Lb\rangle = \langle (L^*a)^*, b\rangle \tag{7.1}$$

where $\langle\, ,\, \rangle$ denotes the dot product. Remember that the dot product of two functions $f, g \in \mathbb{L}_2$ is $\int fg^*\, dt$ while the dot product of two vectors $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{x}^H\mathbf{y}$. Notice that for vectors eq. (7.1) becomes $\mathbf{x}^H\mathbf{L}\mathbf{y} = (\mathbf{L}^H\mathbf{y})^H\mathbf{y}$ which is obviously true. The lhs of eq. (7.1) is computed using L, while the rhs is computed using the adjoint L^*. For the dot-product test, one just needs to load the vectors `x` and `y` with random numbers and perform the two computations. If the two results are not equal (within machine precision), then the computation of either L or L^* is erroneous. Note that truncation errors have identical effects on both operators, so the two results should be almost equal. The dot-product test (for real operators only) is implemented by `sf_dot_test`.

7.1.5 Implementation of operators

It should be evident by now that the implementation of an operator L should have at least four arguments: a variable `x` from which the operand (entity on which L is applied) x is read along with its length `nx`, and the variable `y` in which the result $y = Lx$ is stored and its length n_y.

Also, since every operator comes along with its adjoint, the implementation of the linear operators described later in this chapter, gives also the possibility to compute the adjoint operator. This is done through the boolean `adj` input argument. *When* `adj` *is* `true`*, the adjoint operator L^* computed.* As discussed before, the domains of x and Lx are in general different, therefore L^* cannot be applied on x. However it can always be applied on Lx or some y, which has the same domain as Lx. For this reason, when `adj` is `true`, the operand is y and the result is x and thus, `y` is used as input and the result is stored in `x`. As an example if `sf_copy_lop` (the identity operator) is called, then the result is that $y \leftarrow x$. However if additionally `adj` is `true`, then the result will be $x \leftarrow y$. If `adjnull` (the null operator) is called, then the result is that $y \leftarrow 0$. However if additionally `adj` is `true`, then the result will be $x \leftarrow 0$.

Finally, it is often the case that we need to compute $y \leftarrow Lx$ but $y \leftarrow y + Lx$. For this reason another boolean argument, namely `add` is defined. If `add` is true, then $y \leftarrow y + Lx$. Considering the same example with the identity operator, if `sf_copy_lop` is called with `add` being `true`, then $y \leftarrow y + x$. If additionally `adj` is `true`, then $x \leftarrow y + x$. Or if `adjnull` is called with `add` being `true`, if `adj` is `false`, $y \leftarrow y$ and if `adj` is `true`, then $x \leftarrow x$ (so in essence, if `add` is `true`, no matter what the value of `adj`, nothing happens).

As a conclusion, the linear operators described in this chapter have all the following form:

`oper(adj, add, nx, ny, x, y)`,

where `adj` and `add` are boolean, `nx` and `ny` are integers and `x` and `y` are pointers of various but the same data type. Table 7.1 summarizes the effect of the `adj` and `add` variables.

Table 7.1: Returned values for linear operations.

`adj`	`add`	description	returns
0	0	normal operation	$y \leftarrow Lx$
0	1	normal operation with addition	$y \leftarrow y + Lx$
1	0	adjoint operation	$x \leftarrow L^*y$
1	1	adjoint operation with addition	$x \leftarrow x + L^*y$

7.2 Adjoint zeroing (adjnull.c)

The null operator is defined by

$$y = 0x = 0, \qquad \text{with} \quad y_t \leftarrow 0.$$

Its adjoint is

$$x = 0^*y = 0, \qquad \text{with} \quad x_t \leftarrow 0.$$

7.2.1 sf_adjnull

Call

```
sf_adjnull(adj, add, nx, ny, x, y);
```

Definition

```
void sf_adjnull (bool adj /* adjoint flag */,
                 bool add /* addition flag */,
                 int nx   /* size of \texttt{x} */,
                 int ny   /* size of \texttt{y} */,
                 float* x,
                 float* y)
/*< Zeros out the output (unless add is true).
    Useful first step for any linear operator. >*/
{
   ...
}
```

Input parameters

`adj` adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow 0^* y$ or $x \leftarrow x + 0^* y$.

`add` addition flag (`bool`). If `true`, then $y \leftarrow y + 0x$ or $x \leftarrow x + 0^* y$.

`nx` size of `x` (`int`).

`ny` size of `y` (`int`).

`x` input data or output (`float*`).

`y` output or input data (`float*`).

7.2.2 sf_cadjnull

The same as `sf_adjnull` but for complex data.

Call

```
sf_cadjnull(adj, add, nx, ny, x, y);
```

Definition

```
void sf_cadjnull (bool adj /* adjoint flag */,
                  bool add /* addition flag */,
                  int nx   /* size of \texttt{x} */,
                  int ny   /* size of \texttt{y} */,
                  sf_complex* x,
                  sf_complex* y)
/*< adjnull version for complex data. >*/
{
   ...
}
```

Input parameters

`adj` adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow 0^* y$ or $x \leftarrow x + 0^* y$.

`add` addition flag (`bool`). If `true`, then $y \leftarrow y + 0x$ or $x \leftarrow x + 0^* y$.

`nx` size of `x` (`int`).

`ny`	size of `y` (`int`).
`x`	input data or output (`sf_complex*`).
`y`	output or input data (`sf_complex*`).

7.3 Simple identity (copy) operator (copy.c)

The identity operator is defined by

$$y = 1x = x, \qquad \text{with} \quad y_t \leftarrow x_t .$$

Its adjoint is

$$x = 1^* y = y, \qquad \text{with} \quad x_t \leftarrow y_t .$$

7.3.1 sf_copy_lop

Call

```
sf_copy_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_copy_lop (bool adj, bool add, int nx, int ny, float* xx, float* yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow 1^* y$ or $x \leftarrow x + 1^* y$.
`add`	addition flag (`bool`). If `true`, then $y \leftarrow y + 1x$ or $x \leftarrow x + 1^* y$.
`nx`	size of `x` (`int`). `nx` must equal `ny`.
`ny`	size of `y` (`int`). `ny` must equal `nx`.
`x`	input data or output (`float*`).
`y`	output or input data (`float*`).

7.4 Simple identity (copy) operator for complex data (ccopy.c)

This is the same operator as `sf_copy_lop` but for complex data. In particular, the identity operator is defined by

$$y = 1x = x, \qquad \text{with} \quad y_t \leftarrow x_t .$$

Its adjoint is

$$x = 1^* y = y, \qquad \text{with} \quad x_t \leftarrow y_t .$$

7.4.1 sf_ccopy_lop

Call

```
sf_ccopy_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_ccopy_lop (bool adj, bool add, int nx, int ny,
                   sf_complex* xx, sf_complex* yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow 1^* y$ or $x \leftarrow x + 1^* y$.
`add`	addition flag (`bool`). If `true`, then $y \leftarrow y + 1x$ or $x \leftarrow x + 1^* y$.
`nx`	size of `x` (`int`). `nx` must equal `ny`.
`ny`	size of `y` (`int`). `ny` must equal `nx`.
`x`	input data or output (`sf_complex*`).
`y`	output or input data (`sf_complex*`).

7.5 Simple mask operator (mask.c)

This mask operator is defined by

$$y = L_m x = mx, \qquad \text{with} \quad y_t \leftarrow m_t x_t,$$

where m_t takes binary values, i.e. $m_t = 0$ or 1. Its adjoint is

$$x = L_m^* y = my, \qquad \text{with} \quad x_t \leftarrow m_t y_t,$$

7.5.1 sf_mask_init

Initializes the static variable `m` with boolean values, to be used in the `sf_mask_lop` or `sf_cmask_lop`.

Call

```
sf_mask_init (m);
```

Definition

```
void sf_mask_init(const bool *m)
/*< initialize with mask >*/
{
   ...
}
```

Input parameters

`m` a pointer to boolean values (`const bool*`).

7.5.2 sf_mask_lop

Call

```
sf_mask_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_mask_lop(bool adj, bool add, int nx, int ny, float *x, float *y)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj` adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow L_m^* y$ or $x \leftarrow x + L_m^* y$.

`add` addition flag (`bool`). If `true`, then $y \leftarrow y + L_m x$ or $x \leftarrow x + L_m^* y$.

`nx` size of `x` (`int`). `nx` must equal `ny`.

`ny` size of `y` (`int`). `ny` must equal `nx`.

`x` input data or output (`float*`).

`y` output or input data (`float*`).

7.5.3 sf_cmask_lop

The same as `sf_mask_lop` but for complex data.

Call

```
sf_cmask_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_cmask_lop(bool adj, bool add, int nx, int ny,
                  sf_complex *x, sf_complex *y)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj` adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow L_m^* y$ or $x \leftarrow x + L_m^* y$.

`add` addition flag (`bool`). If `true`, then $y \leftarrow y + L_m x$ or $x \leftarrow x + L_m^* y$.

`nx` size of `x` (`int`). `nx` must equal `ny`.

`ny`	size of `y` (`int`). `ny` must equal `nx`.
`x`	input data or output (`sf_complex*`).
`y`	output or input data (`sf_complex*`).

7.6 Simple weight operator (weight.c)

This weight operator is defined by

$$y = L_w x = wx, \qquad \text{with} \quad y_t \leftarrow w_t x_t.$$

Its adjoint is

$$x = L_w^* y = wy, \qquad \text{with} \quad x_t \leftarrow w_t y_t.$$

Note that for complex data the weight w must still be real.

There is also an in-place ($x \leftarrow L_w x$) version of the operator, which multiplies the input data with the square of w i.e.

$$x = L_w x = w^2 x, \qquad \text{with} \quad x_t \leftarrow w_t^2 x_t.$$

7.6.1 sf_weight_init

Initializes the weights to be applied as linear operator, by assigning value to a static parameter.

Call

```
sf_weight_init(w);
```

Definition

```
void sf_weight_init(float *w1)
/*< initialize >*/
{
    ..
}
```

Input parameters

`w`	values of the weights (`float*`).

7.6.2 sf_weight_lop

Applies the linear operator with the weights initialized by `sf_weight_init`.

Call

```
sf_weight_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_weight_lop (bool adj, bool add, int nx, int ny, float* xx, float* yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow L_w^* y$ or $x \leftarrow x + L_w^* y$.
`add`	addition flag (`bool`). If `true`, then $y \leftarrow y + L_w x$ or $x \leftarrow x + L_w^* y$.
`nx`	size of `x` (`int`). `nx` must equal `ny`.
`ny`	size of `y` (`int`). `ny` must equal `nx`.
`x`	input data or output (`float*`).
`y`	output or input data (`float*`).

7.6.3 sf_cweight_lop

The same as `sf_weight_lop` but for complex data.

Call

```
sf_cweight_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_cweight_lop (bool adj, bool add, int nx, int ny,
                     sf_complex* xx, sf_complex* yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow L_w^* y$ or $x \leftarrow x + L_w^* y$.
`add`	addition flag (`bool`). If `true`, then $y \leftarrow y + L_w x$ or $x \leftarrow x + L_w^* y$.
`nx`	size of `x` (`int`). `nx` must equal `ny`.
`ny`	size of `y` (`int`). `ny` must equal `nx`.
`x`	input data or output (`sf_complex*`).
`y`	output or input data (`sf_complex*`).

7.6.4 sf_weight_apply

Creates a product of the weights squared and the input `x`.

Call

```
sf_weight_apply (nx, x);
```

Definition

```
void sf_weight_apply(int nx, float *xx)
/*< apply weighting in place >*/
{
   ...
}
```

Input parameters

`nx`	size of `x` (`int`).
`x`	input data and output (`float*`).

7.6.5 sf_cweight_apply

The same as the `sf_weight_apply` but for the complex numbers.

Call

```
sf_cweight_apply (nx, x);
```

Definition

```
void sf_cweight_apply(int nx, sf_complex *xx)
/*< apply weighting in place >*/
{
   ...
}
```

Input parameters

`nx`	size of `x` (`int`).
`x`	input data and output (`sf_complex*`).

7.7 1-D finite difference (igrad1.c)

The 1-D finite difference operator is defined by

$$y = Dx, \qquad \text{with} \quad y_t \leftarrow x_{t+1} - x_t.$$

Its adjoint is

$$x = D^* y, \qquad \text{with} \quad x_t \leftarrow -(y_t - y_{t-1}),\ x_0 = -y_0.$$

7.7.1 sf_igrad1_lop

Call

```
sf_igrad1_lop(adj, add, nx, ny, x, y);
```

Definition

```
void  sf_igrad1_lop(bool adj, bool add,
                    int nx, int ny, float *xx, float *yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow D^* y$ or $x \leftarrow x + D^* y$.
`add`	addition flag (`bool`). If `true`, then $y \leftarrow y + Dx$ or $x \leftarrow x + D^* y$.
`nx`	size of `x` (`int`).
`ny`	size of `y` (`int`).
`x`	input data or output (`float*`).
`y`	output or input data (`float*`).

7.8 Causal integration (causint.c)

This causal integration operator is defined by

$$y = Lx, \qquad \text{with} \quad y_t \leftarrow \sum_{\tau=0}^{t} x_\tau .$$

Its adjoint is

$$x = L^* y, \qquad \text{with} \quad x_t \leftarrow \sum_{\tau=t}^{T-1} y_\tau ,$$

where T is the total number of samples of x.

7.8.1 sf_causint_lop

Call

```
sf_causint_lop (adj, add, nx, ny, x, y);
```

Defintion

```
sf_causint_lop (adj, add, nx, ny, x, y)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $x \leftarrow L^* y$ or $x \leftarrow x + L^* y$.
`add`	addition flag (`bool`). If `true`, then $y \leftarrow y + Lx$ or $x \leftarrow x + L^* y$.
`nx`	size of `x` (`int`).
`ny`	size of `y` (`int`).
`x`	input data or output (`float*`).
`y`	output or input data (`float*`).

7.9 Chaining linear operators (chain.c)

Calculates products of operators.

7.9.1 sf_chain

Chains two operators L_1 and L_2:

$$d = (L_2 L_1) m.$$

Its adjoint is

$$m = (L_2 L_1)^* d = L_1^* L_2^* d.$$

Call

```
sf_chain (oper1,oper2, adj,add, nm,nd,nt, mod,dat,tmp);
```

Definition

```
void sf_chain( sf_operator oper1     /* outer operator */,
               sf_operator oper2     /* inner operator */,
               bool adj              /* adjoint flag */,
               bool add              /* addition flag */,
               int nm                /* model size */,
               int nd                /* data size */,
               int nt                /* intermediate size */,
               /*@out@*/ float* mod  /* [nm] model */,
               /*@out@*/ float* dat  /* [nd] data */,
               float* tmp            /* [nt] intermediate */)
/*< Chains two operators, computing oper1{oper2{mod}}
  or its adjoint. The tmp array is used for temporary storage. >*/
{
   ...
}
```

Input parameters

`oper1`	outer operator, L_1 (`sf_operator`).
`oper2`	inner operator, L_2 (`sf_operator`).
`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $m \leftarrow (L_2L_1)^*d$ or $m \leftarrow m + (L_2L_1)^*d$.
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d+(L_2L_1)m$ or $m \leftarrow m+(L_2L_1)^*d$ is computed.
`nm`	size of the model `mod` (`int`).
`nd`	size of the data `dat` (`int`).
`nt`	size of the intermediate result `tmp` (`int`).
`mod`	the model, m (`float*`).
`dat`	the data, d (`float*`).
`tmp`	intermediate result (`float*`).

7.9.2 sf_cchain

The same as `sf_chain` but for complex data.

Call

```
sf_cchain (oper1,oper2, adj,add, nm,nd,nt, mod, dat, tmp);
```

Definition

```
void sf_cchain( sf_coperator oper1         /* outer operator */,
                sf_coperator oper2         /* inner operator */,
                bool adj                   /* adjoint flag */,
                bool add                   /* addition flag */,
                int nm                     /* model size */,
                int nd                     /* data size */,
                int nt                     /* intermediate size */,
                /*@out@*/ sf_complex* mod  /* [nm] model */,
                /*@out@*/ sf_complex* dat  /* [nd] data */,
                sf_complex* tmp            /* [nt] intermediate */)
/*< Chains two complex operators, computing oper1{oper2{mod}}
    or its adjoint. The tmp array is used for temporary storage. >*/
{
   ...
}
```

Input parameters

`oper1`	outer operator, L_1 (`sf_coperator`).
`oper2`	inner operator, L_2 (`sf_coperator`).
`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $m \leftarrow (L_2L_1)^*d$ or $m \leftarrow m + (L_2L_1)^*d$.
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d+(L_2L_1)m$ or $m \leftarrow m+(L_2L_1)^*d$ is computed.

`nm`	size of the model `mod` (`int`).
`nd`	size of the data `dat` (`int`).
`nt`	size of the intermediate result `tmp` (`int`).
`mod`	the model, m (`sf_complex*`).
`dat`	the data, d (`sf_complex*`).
`tmp`	intermediate result (`sf_complex*`).
`tmp`	the intermediate storage (`sf_complex*`).

7.9.3 sf_array

For two operators L_1 and L_2, it calculates:

$$d = Lm,$$

or its adjoint

$$m = L^* d,$$

where

$$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \quad \text{and} \quad d = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

Call

```
sf_array (oper1,oper2, adj,add, nm,nd1,nd2, mod,dat1,dat2);
```

Definition

```
void sf_array( sf_operator oper1     /* top operator */,
               sf_operator oper2     /* bottom operator */,
               bool adj              /* adjoint flag */,
               bool add              /* addition flag */,
               int nm                /* model size */,
               int nd1               /* top data size */,
               int nd2               /* bottom data size */,
               /*@out@*/ float* mod  /* [nm] model */,
               /*@out@*/ float* dat1 /* [nd1] top data */,
               /*@out@*/ float* dat2 /* [nd2] bottom data */)
/*< Constructs an array of two operators,
    computing {oper1{mod},oper2{mod}} or its adjoint. >*/
{
   ...
}
```

Input parameters

`oper1`	top operator, L_1 (`sf_operator`).
`oper2`	bottom operator, L_2 (`sf_operator`).
`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $m \leftarrow L^* d$ or $m \leftarrow m + L^* d$.
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d + Lm$ or $m \leftarrow m + L^* d$ is computed.
`nm`	size of the model, `mod` (`int`).
`nd1`	size of the top data, `dat1 dat1` (`int`).
`nd2`	size of the bottom data, `dat2` (`int`).
`mod`	the model, m (`float*`).
`dat1`	the top data, d_1 (`float*`).
`dat2`	the bottom data, d_2 (`float*`).

7.9.4 sf_normal

Applies a normal operator (self-adjoint) to the model, i.e. it calculates

$$d = LL^* m.$$

Call

```
sf_normal (oper, add, nm,nd, mod,dat,tmp);
```

Definition

```
void sf_normal (sf_operator oper /* operator */,
                bool add         /* addition flag */,
                int nm           /* model size */,
                int nd           /* data size */,
                float *mod       /* [nd] model */,
                float *dat       /* [nd] data */,
                float *tmp       /* [nm] intermediate */)
/*< Applies a normal operator (self-adjoint) >*/
{
   ...
}
```

Input parameters

`oper`	the operator, L (`sf_operator`).
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d + LL^* m$.
`nm`	size of the model, `mod` (`int`).
`nd`	size of the data, `dat` (`int`).
`mod`	the model, m (`float*`).
`dat`	the data, d (`float*`).
`tmp`	the intermediate result (`float*`).

7.9.5 sf_chain3

Chains three operators L_1, L_2 and L_3:

$$d = (L_3 L_2 L_1) m.$$

Its adjoint is

$$m = (L_3 L_2 L_1)^* d = L_1^* L_2^* L_3^* d.$$

Call

```
sf_chain3 (oper1,oper2,oper3, adj,add, nm,nt1,nt2,nd, mod,dat,tmp1,tmp2);
```

Definition

```
void sf_chain3 (sf_operator oper1 /* outer operator */,
                sf_operator oper2 /* middle operator */,
                sf_operator oper3 /* inner operator */,
                bool adj          /* adjoint flag */,
                bool add          /* addition flag */,
                int nm            /* model size */,
                int nt1           /* inner intermediate size */,
                int nt2           /* outer intermediate size */,
                int nd            /* data size */,
                float* mod        /* [nm] model */,
                float* dat        /* [nd] data */,
                float* tmp1       /* [nt1] inner intermediate */,
                float* tmp2       /* [nt2] outer intermediate */)
/*< Chains three operators, computing oper1{oper2{poer3{{mod}}} or its adjoint.
  The tmp1 and tmp2 arrays are used for temporary storage. >*/
{
   ...
}
```

Input parameters

`oper1`	outer operator (`sf_operator`).
`oper2`	middle operator (`sf_operator`).
`oper3`	inner operator (`sf_operator`).
`adj`	adjoint flag (`bool`). If `true`, then the adjoint is computed, i.e. $m \leftarrow L_1^* L_2^* L_3^* d$ or $m \leftarrow m + L_1^* L_2^* L_3^* d$.
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d + L_3 L_2 L_1 m$ or $m \leftarrow m + L_1^* L_2^* L_3^* d$.
`nm`	size of the model, `mod` (`int`).
`nt1`	inner intermediate size (`int`).
`nt2`	outer intermediate size (`int`).
`ny`	size of the data, `dat` (`int`).
`mod`	the model, x (`float*`).

`dat`	the data, d (`float*`).
`tmp1`	the inner intermediate result (`float*`).
`tmp2`	the outer intermediate result (`float*`).

7.10 Dot product test for linear operators (dottest.c)

Performs the dot product test (see p. 117), to check whether the adjoint of the operator is coded incorrectly. Coding is incorrect if any of

$$\langle Lm_1, d_2 \rangle = \langle m_1, L^* d_2 \rangle \quad \text{or} \quad \langle d_1 + Lm_1, d_2 \rangle = \langle m_1, m_2 + L^* d_2 \rangle$$

does not hold (within machine precision). m_1 and d_2 are random vectors.

7.10.1 sf_dot_test

`dot1[0]` must equal `dot1[1]` and `dot2[0]` must equal `dot2[1]` for the test to pass.

Call

```
sf_dot_test (oper, nm, nd, dot1, dot2);
```

Definition

```
void sf_dot_test(sf_operator oper /* linear operator */,
                 int nm           /* model size */,
                 int nd           /* data size */,
                 float* dot1      /* first output */,
                 float* dot2      /* second output */)
{
   ...
}
```

Input parameters

`oper`	the linear operator, whose adjoint is to be tested (`sf_operator`).
`nm`	size of the models (`int`).
`nd`	size of the data (`int`).
`dot1`	first output dot product (`float*`).
`dot2`	second output dot product (`float*`).

Chapter 8

Data analysis

8.1 FFT (kiss_fftr.c)

8.1.1 sf_kiss_fftr_alloc

Allocates the memory for the FFT and returns an object of type `kiss_fftr_cfg`.

Call

```
kiss_fftr_alloc(nfft, inverse_fft, mem, lenmem);
```

Definition

```
kiss_fftr_cfg kiss_fftr_alloc(int nfft,int inverse_fft,void * mem,size_t * lenmem)
{
   ...
}
```

Input parameters

`nfft`	length of the forward FFT (`int`).
`inverse_fft`	length of the inverse FFT (`int`).
`mem`	pointer to the memory allocated for FFT (`void*`).
`lenmem`	size of the allocated memory (`size_t`).

Input parameters

`st`	an object for FFT (`kiss_fftr_cfg`).

8.1.2 kiss_fftr

Performs the forward FFT on the input `timedata` which is real, and stores the transformed complex `freqdata` in the location specified in the input.

Call

```
kiss_fftr( st, timedata, freqdata)
```

Definition

```
void kiss_fftr(kiss_fftr_cfg st,const kiss_fft_scalar *timedata,kiss_fft_cpx *fr
eqdata)
{
    /* input buffer timedata is stored row-wise */

    /* The real part of the DC element of the frequency spectrum in st->tmpbuf
     * contains the sum of the even-numbered elements of the input time sequence
     * The imag part is the sum of the odd-numbered elements
     *
     * The sum of tdc.r and tdc.i is the sum of the input time sequence.
     *      yielding DC of input time sequence
     * The difference of tdc.r - tdc.i is the sum of the input (dot product) [1,-1,1,-1...
     *      yielding Nyquist bin of input time sequence
     */

   ...
}
```

Input parameters

`st`	an object for the forward FFT (`kiss_fftr_cfg`).
`timedata`	time data which is to be transformed (`const kiss_fft_scalar*`).
`freqdata`	location where the transformed data is to be stored (`kiss_fft_cpx*`).

8.1.3 kiss_fftri

Performs the inverse FFT on the input `timedata` which is real, and stores the transformed complex `freqdata` in the location specified in the input.

Call

```
kiss_fftri(st, freqdata, timedata);
```

Definition

```
void kiss_fftri(kiss_fftr_cfg st,const kiss_fft_cpx *freqdata,kiss_fft_scalar *t
imedata)
/* input buffer timedata is stored row-wise */
{
   ...
}
```

Input parameters

`st`	an object for the inverse FFT (`kiss_fftr_cfg`).
`timedata`	location where the inverse time data is to be stored (`const kiss_fft_scalar*`).
`freqdata`	complex frequency data which is to be inverse transformed (`kiss_fft_cpx*`).

8.2 Cosine window weighting function (tent2.c)

8.2.1 sf_tent2

Sets the weights for the windows defined for each dimension.

Call

```
sf_tent2 (dim, nwind, windwt);
```

Definition

```
void sf_tent2 (int dim          /* number of dimensions */,
               const int* nwind /* window size [dim] */,
               float* windwt    /* window weight */)
/*< define window weight >*/
{
   ...
}
```

Input parameters

`dim`	number of dimensions (`int`).
`nwind`	window size [dim] (`const int`).
`windwt`	window weight (`const int`).

8.3 Anisotropic diffusion, 2-D (impl2.c)

8.3.1 sf_impl2_init

Initializes the required variables and allocates the required space for the anisotropic diffusion.

Call

```
void sf_impl2_init (r1,r2, n1_in,n2_in, tau, pclip,
                    up_in, verb_in, dist_in, nsnap_in, snap_in);
```

Definition

```
void sf_impl2_init (float r1, float r2   /* radius */,
                 int n1_in, int n2_in /* data size */,
                 float tau            /* duration */,
                 float pclip          /* percentage clip */,
                 bool up_in           /* weighting case */,
                 bool verb_in         /* verbosity flag */,
                 float *dist_in       /* optional distance function */,
                 int nsnap_in         /* number of snapshots */,
                 sf_file snap_in      /* snapshot file */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`r1` radius in the first dimension (`float`).

`r2` radius in the second dimension (`float`).

`n1_in` length of first dimension in the input data (`int`).

`n2_in` length of second dimension in the input data (`int`).

`tau` duration (`float`).

`pclip` percentage clip (`float`).

`up_in` weighting case (`bool`).

`verb_in` verbosity flag (`bool`).

`dist_in` optional distance function (`float*`).

`nsnap_in` number of snapshots (`int`).

`snap_in` snapshot file (`sf_file`).

8.3.2 sf_impl2_close

Frees the space allocated by `sf_impl2_init`.

Definition

```
sf_impl2_close ();
```

Definition

```
void sf_impl2_close (void)
/*< free allocated storage >*/
{
   ...
}
```

8.3.3 sf_impl2_set

Computes the weighting function for the anisotropic diffusion.

Call

```
sf_impl2_set(x);
```

Definition

```
void sf_impl2_set(float ** x)
/*< compute weighting function >*/
{
   ...
}
```

Input parameters

`x`	data (`float**`).

8.3.4 sf_impl2_set

Applies the anisotropic diffusion.

Call

```
sf_impl2_apply (x, set, adj);
```

Definition

```
void sf_impl2_apply (float **x, bool set, bool adj)
/*< apply diffusion >*/
{
   ...
}
```

Input parameters

`x`	data (`float*`).
`set`	whether the weighting function needs to be computed (`bool`).
`adj`	whether the weighting function needs to be applied (`bool`).

8.3.5 sf_impl2_lop

Applies either `x` or `y` as linear operator to `y` or `x` and output `x` or `y`, depending on whether `adj` is true or false.

Call

```
sf_impl2_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_impl2_lop (bool adj, bool add, int nx, int ny, float* x, float* y)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether the output is `y` or `x` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `x` (`int`).
`ny`	size of `y` (`int`).
`x`	data or operator, depending on whether `adj` is true or false (`sf_complex*`).
`y`	data or operator, depending on whether `adj` is true or false (`sf_complex*`).

Chapter 9

Filtering

9.1 Frequency-domain filtering (freqfilt.c)

9.1.1 sf_freqfilt_init

Initializes the required variables and allocates the required space for frequency filtering.

Call

```
sf_freqfilt_init(nfft1, nw1);
```

Definition

```
void sf_freqfilt_init(int nfft1 /* time samples (possibly padded) */,
                      int nw1   /* frequency samples */)
/*< Initialize >*/
{
   ...
}
```

Input parameters

`nfft1`	number of time samples (`int`).
`nw1`	number of frequency samples (`int`).

9.1.2 sf_freqfilt_set

Initializes a zero phase filter.

Call

```
sf_freqfilt_set(filt);
```

Definition

```
void sf_freqfilt_set(float *filt /* frequency filter [nw] */)
/*< Initialize filter (zero-phase) >*/
{
   ...
}
```

Input parameters

`filt`	the frequency filter (`float*`).

9.1.3 sf_freqfilt_cset

Initializes a non-zero phase filter (filter with complex values).

Call

```
sf_freqfilt_cset (filt);
```

Definition

```
void sf_freqfilt_cset(kiss_fft_cpx *filt /* frequency filter [nw] */)
/*< Initialize filter >*/
{
   ...
}
```

Input parameters

`filt`	the frequency filter. Must be of type `kiss_fft_cpx*`.

9.1.4 sf_freqfilt_close

Frees the space allocated by `sf_freqfilt_init`.

Call

```
sf_freqfilt_close();
```

Definition

```
void sf_freqfilt_close(void)
/*< Free allocated storage >*/
{
   ...
}
```

9.1.5 sf_freqfilt

Applies the frequency filtering to the input data.

Call

```
sf_freqfilt(nx, x);
```

Definition

```
void sf_freqfilt(int nx, float* x)
/*< Filtering in place >*/
{
   ...
}
```

Input parameters

`nx`	data length (`int`).
`x`	the data (`float*`).

9.1.6 sf_freqfilt_lop

Applies the frequency filtering to either `y` or `x`, depending on whether `adj` is true of false and then applies the result to `x` or `y` as a linear operator.

Call

```
sf_freqfilt_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_freqfilt_lop (bool adj, bool add, int nx, int ny, float* x, float* y)
/*< Filtering as linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether frequency filter applied to `y` or `x`. Must be of type `bool`.
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `x` (`int`).
`ny`	size of `y` (`int`).
`x`	data or operator, depending on whether `adj` is true or false (`float*`).
`y`	data or operator, depending on whether `adj` is true or false (`float*`).

9.2 Frequency-domain filtering in 2-D (freqfilt.c)

9.2.1 sf_freqfilt2_init

Initializes the required variables and allocates the required space for frequency filtering for 2D data.

Call

```
sf_freqfilt2_init (n1, n2, nw1);
```

Definition

```
void sf_freqfilt2_init(int n1, int n2 /* data dimensions */,
                       int nw1       /* number of frequencies */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`n1`	number of time samples in first dimension (`int`).
`n2`	number of time samples in second dimension (`int`).
`nw1`	number of frequencies (`int`).

9.2.2 sf_freqfilt2_set

Initializes a zero phase filter.

Call

```
sf_freqfilt2_set (filt);
```

Input parameters

`filt`	the frequency filter (`float**`).

Definition

```
void sf_freqfilt2_set(float **filt)
/*< set the filter >*/
{
   ...
}
```

9.2.3 sf_freqfilt2_close

Frees the space allocated by `sf_freqfilt2_init`.

Call

```
sf_freqfilt2_close();
```

Definition

```
void sf_freqfilt2_close(void)
/*< free allocated storage >*/
{
   ...
}
```

9.2.4 sf_freqfilt2_spec

This function 2D spectrum of the input data.

Call

```
sf_freqfilt2_spec (x, y);
```

Definition

```
void sf_freqfilt2_spec (const float* x /* input */, float** y /* spectrum */)
/*< compute 2-D spectrum >*/
{
   ...
}
```

Input parameters

`x`	the data (`const float*`).
`y`	the data (`float**`).

9.2.5 sf_freqfilt2_lop

Applies the frequency filtering to either `y` or `x`, depending on whether `adj` is true of false and then applies the result to `x` or `y` as a linear operator.

Call

```
sf_freqfilt2_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_freqfilt2_lop (bool adj, bool add, int nx, int ny, float* x, float* y)
/*< linear filtering operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether frequency filter applied to `y` or `x` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `x` (`int`).

`ny`	size of `y` (`int`).
`x`	data or operator, depending on whether `adj` is true or false (`float*`).
`y`	data or operator, depending on whether `adj` is true or false (`float*`).

9.3 Helical convolution (helicon.c)

9.3.1 sf_helicon_init

Initializes an object of type `sf_filter` to be used in the linear operator function.

Call

```
sf_helicon_init(bb);
```

Definition

```
void sf_helicon_init (sf_filter bb)
/*<  Initialized with the filter. >*/
{
   ...
}
```

Input parameters

`bb`	the filter object (`sf_filter`).

9.3.2 sf_helicon_lop

Does the helical convolution. It applies the filter to either `yy` or `xx`, depending on whether `adj` is true of false and then applies the result to `xx` or `yy` as a linear operator.

Call

```
sf_helicon_lop (adj, add, nx, ny, xx, yy);
```

Definition

```
void sf_helicon_lop( bool adj, bool add,
                     int nx, int ny, float* xx, float*yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether the filter is applied to `yy` or `xx` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `xx` (`int`).
`ny`	size of `yy` (`int`).
`xx`	data or operator, depending on whether `adj` is true or false (`float*`).
`yy`	data or operator, depending on whether `adj` is true or false (`float*`).

9.4 Helical filter definition and allocation (helix.c)

9.4.1 sf_allocatehelix

Initializes the filter.

Call

```
aa = sf_allocatehelix(nh);
```

Definition

```
sf_filter sf_allocatehelix( int nh)
/*< allocation >*/
{
   ...
}
```

Input parameters

`nh`	filter length (`int`).

Output

`aa`	object for helix filter. It is of type `sf_filter`.

9.4.2 sf_deallocatehelix

Frees the space allocated by `sf_allocatehelix` for the filter.

Definition

```
sf_deallocatehelix (aa);
```

Call

```
void sf_deallocatehelix( sf_filter aa)
/*< deallocation >*/
{
   ...
}
```

Input parameters

`aa` the filter (`sf_filter`).

9.4.3 sf_displayhelix

Displays the filter.

Call

```
sf_displayhelix(aa);
```

Definition

```
void sf_displayhelix (sf_filter aa)
/*< display filter >*/
{
   ...
}
```

Input parameters

`aa` the filter (`sf_filter`).

9.5 Recursive convolution (polynomial division) (recfilt.c)

9.5.1 sf_recfilt_init

Initializes the linear filter by allocating the required space and initializing the required variables.

Call

```
sf_recfilt_init(nd, nb, bb);
```

Definition

```
void sf_recfilt_init( int nd    /* data size */,
                   int nb    /* filter size */,
                   float* bb /* filter [nb] */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`nd` size of the data which is to be filtered (`int`).

`nb` size of the filter (`int`).

`bb` filter which is to be applied (`float*`).

9.5.2 sf_recfilt_lop

Applies the linear operator to `xx` (or `yy`) and the result is applied to `yy` (or `xx`), depending on whether `adj` is false or true, with the operator initialized by `sf_recfilt_init`.

Call

```
sf_recfilt_lop (adj, add, nx, ny, xx, yy);
```

Definition

```
void sf_recfilt_lop( bool adj, bool add, int nx, int ny, float* xx, float*yy)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether filter is applied to `yy` or `xx` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `xx` (`int`).
`ny`	size of `yy` (`int`).
`xx`	data or operator, depending on whether `adj` is true or false (`float*`).
`yy`	data or operator, depending on whether `adj` is true or false (`float*`).

9.5.3 sf_recfilt_close

Frees the space allocated by `sf_recfilt_init`.

Call

```
sf_recfilt_close ();
```

Definition

```
void sf_recfilt_close (void)
/*< free allocated storage >*/
{
   ...
}
```

9.6 Cosine Fourier transform (cosft.c)

9.6.1 sf_cosft

Makes preparations for the cosine Fourier transform, by allocating the required spaces.

Call

```
sf_cosft_init(n1);
```

Definition

```
void sf_cosft_init(int n1)
/*< initialize >*/
{
   ...
}
```

Input parameters

`n1_in`	length of the input (`int`).

9.6.2 sf_cosft_close

Frees the allocated space.

Call

```
sf_cosft_close();
```

Definition

```
void sf_cosft_close(void)
/*< free allocated storage >*/
{
   ...
}
```

9.6.3 sf_cosft_frw

This function performs the forward cosine Fourier transform.

Call

```
sf_cosft_frw (q, o1, d1);
```

Definition

```
void sf_cosft_frw (float *q /* data */,
                   int o1   /* first sample */,
                   int d1   /* step */)
/*< forward transform >*/
{
   ...
}
```

Input parameters

`q`	input data (`float`).
`o1`	first sample of the input data (`int`).
`d1`	step size (`int`).

9.6.4 sf_cosft_inv

This function performs the forward cosine Fourier transform.

Call

```
sf_cosft_inv (q, o1, d1);
```

Definition

```
void sf_cosft_inv (float *q /* data */,
                   int o1   /* first sample */,
                   int d1   /* step */)
/*< inverse transform >*/
{
   ...
}
```

Input parameters

`q`	input data (`float`).
`o1`	first sample of the input data (`int`).
`d1`	step size (`int`).

Chapter 10

Solvers

10.1 Banded matrix solver (banded.c)

10.1.1 sf_banded_init

Initializes an object of type `sf_bands` for the banded matrix, that is, it allocates the required spaces and defines initializes the variables.

Call

```
slv = sf_banded_init (n, band);
```

Definition

```
sf_bands sf_banded_init (int n    /* matrix size */,
                          int band /* band size */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`n`	size of the banded matrix (`int`).
`band`	size of the band (`int`).

Output

`slv`	an object of type `sf_bands`.

10.1.2 sf_banded_define

Defines the banded matrix.

Call

```
sf_banded_define (slv, diag, offd);
```

Input parameters

`slv`	an object of type `sf_bands`.
`diag`	diagonal entries in the matrix (`float**`).
`offd`	off-diagonal entries in the matrix (`float**`).

Definition

```
void sf_banded_define (sf_bands slv,
                       float* diag  /* diagonal [n] */,
                       float** offd /* off-diagonal [band][n] */)
/*< define the matrix >*/
{
   ...
}
```

10.1.3 sf_banded_const_define

Defines a banded matrix with constant value in the diagonal.

Call

```
sf_banded_const_define (slv, diag, offd);
```

Definition

```
void sf_banded_const_define (sf_bands slv,
                             float diag        /* diagonal */,
                             const float* offd /* off-diagonal [band] */)
/*< define matrix with constant diagonal coefficients >*/
{
   ...
}
```

Input parameters

`slv`	an object of type `sf_bands`.
`diag`	diagonal entries in the matrix (`float**`).
`offd`	off-diagonal entries in the matrix (`float**`).

10.1.4 sf_banded_const_define_reflect

Defines the banded matrix with constant diagonal values for the reflecting boundary conditions.

Call

```
sf_banded_const_define_reflect (slv, diag, offd);
```

Definition

```
void sf_banded_const_define_reflect (sf_bands slv,
                                     float diag        /* diagonal */,
                                     const float* offd /* off-diagonal [band] */)
/*< define matrix with constant diagonal coefficients
    and reflecting b.c. >*/
{
   ...
}
```

Input parameters

slv   an object of type `sf_bands`.

diag   diagonal entries in the matrix (`float**`).

offd   off-diagonal entries in the matrix (`float**`).

10.1.5 sf_banded_solve

Inverts the banded matrix.

Call

```
sf_banded_solve (slv, b);
```

Definition

```
void sf_banded_solve (const sf_bands slv, float* b)
/*< invert (in place) >*/
{
   ...
}
```

Input parameters

slv   an object of type `sf_bands`. Must be of type `const sf_bands`

b   the inverted matrix values (`float*`).

10.1.6 sf_banded_close

Frees the space allocated for the `sf_bands` object.

Call

```
sf_banded_close (slv);
```

Definition

```
void sf_banded_close (sf_bands slv)
/*< free allocated storage >*/
{
   ...
}
```

Input parameters

`slv`	an object of type `sf_stack`.

10.2 Claerbout's conjugate-gradient iteration (cgstep.c)

10.2.1 sf_cgstep

Evaluates one step of the conjugate gradient method iteration.

Call

```
sf_cgstep(forget, nx, ny, x, g, rr, gg);
```

Definition

```
void sf_cgstep( bool forget     /* restart flag */,
                int nx, int ny  /* model size, data size */,
                float* x        /* current model [nx] */,
                const float* g  /* gradient [nx] */,
                float* rr       /* data residual [ny] */,
                const float* gg /* conjugate gradient [ny] */)
/*< Step of conjugate-gradient iteration. >*/
{
   ...
}
```

Input parameters

`forget`	restart flag (`bool`).
`nx`	size of the model (`int`).
`ny`	size of the data (`int`).
`g`	the gradient (`const float*`).
`rr`	the data residual (`float*`).
`gg`	the conjugate gradient (`const float*`).

Output

`c.r`	real part of the complex number. It is of type `double`.

10.2.2 sf_cgstep_close

Frees the space allocated for the conjugate gradient step calculation.

Call

```
sf_cgstep_close ();
```

Definition

```
void sf_cgstep_close (void)
/*< Free allocated space. >*/
{
   ...
}
```

10.3 Conjugate-gradient with shaping regularization (conjgrad.c)

10.3.1 sf_conjgrad_init

Initializes the conjugate gradient solver by initializing the required variables and allocating the required space.

Call

```
sf_conjgrad_init (np, nx, nd, nr, eps, tol, verb, hasp0);
```

Definition

```
void sf_conjgrad_init(int np1     /* preconditioned size */,
                      int nx1     /* model size */,
                      int nd1     /* data size */,
                      int nr1     /* residual size */,
                      float eps1  /* scaling */,
                      float tol1  /* tolerance */,
                      bool verb1  /* verbosity flag */,
                      bool hasp01 /* if has initial model */)
/*< solver constructor >*/
{
   ...
}
```

Input parameters

`np1`	the size of the preconditioned data (`int`).
`nx1`	size of the model (`int`).
`nd1`	size of the data (`int`).
`nr1`	size of the residual (`int`).

`eps1`	the scaling parameter (`float`).
`tol1`	tolerance to the error in the solution (`float`).
`verb1`	verbosity flag (`bool`).
`hasp01`	if there is a initial model (`bool`).

10.3.2 sf_conjgrad_close

Frees the space allocated for the conjugate gradient solver by `sf_conjgrad_init`.

Definition

```
sf_conjgrad_close();
```

Definition

```
void sf_conjgrad_close(void)
/*< Free allocated space >*/
{
   ...
}
```

10.3.3 sf_conjgrad

Applies the conjugate gradient solver with the shaping filter to the input data.

Definition

```
sf_conjgrad (prec, oper, shape, p, x, dat, niter);
```

Definition

```
void sf_conjgrad(sf_operator prec  /* data preconditioning */,
                 sf_operator oper  /* linear operator */,
                 sf_operator shape /* shaping operator */,
                 float* p          /* preconditioned model */,
                 float* x          /* estimated model */,
                 float* dat        /* data */,
                 int niter         /* number of iterations */)
/*< Conjugate gradient solver with shaping >*/
{
   ...
}
```

Input parameters

`prec`	preconditioning operator (`sf_operator`).
`oper`	the operator (`sf_operator`).
`shape`	the shaping operator (`sf_operator`).
`p`	the preconditioned model (`float*`).

`x`	estimated model (`float*`).
`dat`	the data (`float*`).
`niter`	number of iterations (`int`).

10.4 Conjugate-gradient with preconditioning (conjprec.c)

10.4.1 sf_conjprec_init

Initializes the conjugate gradient solver, that is, it sets the required variables and allocates the required memory.

Call

```
sf_conjprec_init(int nx, nr, eps, tol, verb, hasp0;
```

Definition

```
void sf_conjprec_init(int nx     /* preconditioned size */,
                      int nr     /* residual size */,
                      float eps  /* scaling */,
                      float tol  /* tolerance */,
                      bool verb  /* verbosity flag */,
                      bool hasp0 /* if has initial model */)
/*< solver constructor >*/
{
   ...
}
```

Input parameters

`nx1`	the size of the preconditioned data (`int`).
`nr1`	size of the residual (`int`).
`eps`	the scaling parameter (`float`).
`tol1`	tolerance to the error in the solution (`float`).
`verb1`	verbosity flag (`bool`).
`hasp01`	if there is a initial model (`bool`).

10.4.2 sf_conjprec_close

Frees the allocated space for the conjugate gradient solver.

Call

```
sf_conjprec_close();
```

Definition

```
void sf_conjprec_close(void)
/*< Free allocated space >*/
{
   ...
}
```

10.4.3 sf_conjprec

Applies the conjugate gradient method after preconditioning to the input data.

Call

```
void sf_conjprec(oper, prec, p, x, dat, niter);
```

Definition

```
void sf_conjprec(sf_operator oper  /* linear operator */,
                 sf_operator2 prec /* preconditioning */,
                 float* p          /* preconditioned */,
                 float* x          /* model */,
                 const float* dat  /* data */,
                 int niter         /* number of iterations */)
/*< Conjugate gradient solver with preconditioning >*/
{
   ...
}
```

Input parameters

`oper`	the operator (`sf_operator`).
`prec`	preconditioning operator (`sf_operator2`).
`p`	the preconditioned data (`float*`).
`x`	model (`float*`).
`dat`	the data (`const float*`).
`niter`	number of iterations (`int`).

10.5 Claerbout's conjugate-gradient iteration for complex numbers (cgstep.c)

10.5.1 sf_ccgstep

Evaluates one step of the Claerbout's conjugate-gradient iteration for complex numbers.

Call

```
sf_ccgstep(forget, nx, ny, x, g, rr, gg);
```

Definition

```
void sf_ccgstep( bool forget          /* restart flag */,
                 int nx               /* model size */,
                 int ny               /* data size */,
                 sf_complex* x        /* current model [nx] */,
                 const sf_complex* g  /* gradient [nx] */,
                 sf_complex* rr       /* data residual [ny] */,
                 const sf_complex* gg /* conjugate gradient [ny] */)
/*< Step of Claerbout's conjugate-gradient iteration for complex operators.
    The data residual is rr = A x - dat
>*/
{
   ...
}
```

Input parameters

`forget`	restart flag (`bool`).
`nx`	size of the model (`int`).
`ny`	size of the data (`int`).
`x`	current model (`sf_complex*`).
`g`	the gradient. Must be of type `const sf_complex*`.
`rr`	the data residual (`sf_complex*`).
`gg`	the conjugate gradient. Must be of type `const sf_complex*`.

10.5.2 sf_ccgstep_close

Frees the space allocated for `sf_ccgstep`.

Call

```
sf_ccgstep_close();
```

Definition

```
void sf_ccgstep_close (void)
/*< Free allocated space. >*/
{
   ...
}
```

10.5.3 dotprod

Returns the dot product of two complex numbers or the sum of the dot products if the are two arrays of complex numbers.

Call

```
prod = dotprod (n, x, y);
```

Definition

```
static sf_double_complex dotprod (int n, const sf_complex* x,
                                          const sf_complex* y)
/* complex dot product */
{
   ...
}
```

Input parameters

`n`	size of the array of complex numbers (`int`).
`x`	a complex number (`sf_complex*`).
`y`	a complex number (`sf_complex*`).

Output

`prod`	dot product of the complex numbers. It is of type `static sf_double_complex`.

10.6 Conjugate-gradient with shaping regularization for complex numbers (cconjgrad.c)

10.6.1 norm

Returns the L_2 norm of the complex number with double-precision, or the sum of L_2 norms, if there is an array of complex numbers.

Call

```
prod = norm (n, x);
```

Definition

```
static double norm (int n, const sf_complex* x)
/* double-precision L2 norm of a complex number */
{
   ...
}
```

Input parameters

`n`	size of the array of complex numbers (`int`).
`x`	a complex number (`sf_complex*`).

Output

`prod`	L_2 norm of the complex number. It is of type `static double`.

10.6.2 sf_cconjgrad_init

Initializes the complex conjugate gradient solver by initializing the required variables and allocating the required space.

Definition

```
sf_cconjgrad_init (np, nx, nd, nr, eps, tol, verb, hasp0);
```

Definition

```
void sf_cconjgrad_init(int np     /* preconditioned size */,
                       int nx     /* model size */,
                       int nd     /* data size */,
                       int nr     /* residual size */,
                       float eps  /* scaling */,
                       float tol  /* tolerance */,
                       bool verb  /* verbosity flag */,
                       bool hasp0 /* if has initial model */)
/*< solver constructor >*/
{
   ...
}
```

Input parameters

`np`	the size of the preconditioned data (`int`).
`nx`	size of the model (`int`).
`nd`	size of the data (`int`).
`nr`	size of the residual (`int`).
`eps`	the scaling parameter (`float`).
`tol`	tolerance to the error in the solution (`float`).
`verb`	verbosity flag (`bool`).
`hasp0`	if there is a initial model (`bool`).

10.6.3 sf_cconjgrad_close

Frees the space allocated for the complex conjugate gradient solver by `sf_cconjgrad_init`.

Definition

```
sf_cconjgrad_close();
```

Definition

```
void sf_cconjgrad_close(void)
/*< Free allocated space >*/
{
   ...
}
```

10.6.4 sf_cconjgrad

Applies the complex conjugate gradient solver with the shaping filter to the input data.

Definition

```
sf_cconjgrad (prec, oper, shape, p, x, dat, niter);
```

Definition

```
void sf_cconjgrad(sf_coperator prec     /* data preconditioning */,
                  sf_coperator oper     /* linear operator */,
                  sf_coperator shape    /* shaping operator */,
                  sf_complex* p         /* preconditioned model */,
                  sf_complex* x         /* estimated model */,
                  const sf_complex* dat /* data */,
                  int niter             /* number of iterations */)
/*< Conjugate gradient solver with shaping >*/
{
   ...
}
```

Input parameters

`prec`	preconditioning operator (`sf_coperator`).
`oper`	the operator (`sf_coperator`).
`shape`	the shaping operator (`sf_coperator`).
`p`	the preconditioned model (`sf_complex*`).
`x`	estimated model (`sf_complex*`).
`dat`	the data (`sf_complex*`).
`niter`	number of iterations (`int`).

10.7 Conjugate-direction iteration (cdstep.c)

10.7.1 sf_cdstep_init

Creates a list for internal storage.

Call

```
sf_cdstep_init();
```

Definition

```
void sf_cdstep_init(void)
/*< initialize internal storage >*/
{
   ...
}
```

10.7.2 sf_cdstep_close

Frees the space allocated for internal storage by `sf_cdstep_init`.

Call

```
sf_cdstep_close();
```

Definition

```
void sf_cdstep_close(void)
/*< free internal storage >*/
{
   ...
}
```

10.7.3 sf_cdstep

Calculates one step for the conjugate direction iteration, that is, it calculates the new conjugate gradient for the new line search direction.

Call

```
sf_cdstep(forget, nx, ny, x, g, rr, gg);
```

Definition

```
void sf_cdstep(bool forget     /* restart flag */,
               int nx          /* model size */,
               int ny          /* data size */,
               float* x        /* current model [nx] */,
               const float* g  /* gradient [nx] */,
               float* rr       /* data residual [ny] */,
               const float* gg /* conjugate gradient [ny] */)
/*< Step of conjugate-direction iteration.
  The data residual is rr = A x - dat
>*/
{
   ...
}
```

Input parameters

`forget`	restart flag (`bool`).
`nx`	model size (`int`).
`ny`	data size (`int`).
`x`	current model (`float*`).
`g`	gradient (`const float*`).
`rr`	data residual (`float*`).
`gg`	conjugate gradient (`const float*`).

10.7.4 sf_cdstep_diag

Calculates the diagonal of the model resolution matrix.

Call

```
sf_cdstep_diag(nx, res);
```

Definition

```
void sf_cdstep_diag(int nx, float *res /* [nx] */)
/*< compute diagonal of the model resolution matrix >*/
{
   ...
}
```

Input parameters

`nx`	model size (`int`).
`res`	diagonal entries of the model resolution matrix (`float*`).

10.7.5 sf_cdstep_mat

Calculates the complete model resolution matrix.

Call

```
sf_cdstep_mat (nx, res);
```

Definition

```
void sf_cdstep_mat (int nx, float **res /* [nx][nx] */)
/*< compute complete model resolution matrix >*/
{
   ...
}
```

Input parameters

`nx` model size (`int`).

`res` diagonal entries of the model resolution matrix (`float**`).

10.8 Linked list for use in conjugate-direction-type methods (llist.c)

10.8.1 sf_list_init

Creates an empty list. It returns a pointer to the list.

Call

```
l = sf_llist_init();
```

Definition

```
sf_list sf_llist_init(void)
/*< create an empty list >*/
{
   ...
}
```

Output

`l` an empty list (`sf_list`).

10.8.2 sf_llist_rewind

Rewinds the list, that is, it makes the pointer to the current position equal to the first entry position.

Call

```
sf_llist_rewind(l);
```

Definition

```
void sf_llist_rewind(sf_list l)
/*< return to the start >*/
{
   ...
}
```

Input parameters

`l` a list (`sf_list`).

10.8.3 sf_llist_depth

Returns the depth (length) of the list.

Call

```
d = sf_llist_depth(l);
```

Definition

```
int sf_llist_depth(sf_list l)
/*< return list depth >*/
{
   ...
}
```

Input parameters

`l`	a list (`sf_list`).

Output

`l->depth`	depth (length) of the list (`int`).

10.8.4 sf_llist_add

Adds an entry to the list.

Call

```
sf_llist_add(l, g, gn);
```

Definition

```
void sf_llist_add(sf_list l, float *g, double gn)
/*< add an entry in the list >*/
{
   ...
}
```

Input parameters

`l`	a list (`sf_list`).
`g`	value which is to be entered in the list (`float*`).
`gn`	name or key of the value which is to be entered in the list (`double`).

10.8.5 sf_llist_down

Extracts an entry from the list.

Call

```
sf_llist_down(l, g, gn);
```

Definition

```
void sf_llist_down(sf_list l, float **g, double *gn)
/*< extract and entry from the list >*/
{
   ...
}
```

Input parameters

`l`	a list (`sf_list`).
`g`	location where extracted value is to be stored (`float**`).
`gn`	location where the name or key of the value is to be stored (`double`).

10.8.6 sf_llist_close

Frees the space allocated for the `sf_list` object (list).

Call

```
sf_llist_close(l);
```

Definition

```
void sf_llist_close(sf_list l)
/*< free allocated storage >*/
{
   ...
}
```

Input parameters

`l`	a list (`sf_list`).

10.8.7 sf_llist_chop

Removes the first entry from the list.

Call

```
sf_llist_chop(l);
```

Definition

```
void sf_llist_chop(sf_list l)
/*< free the top entry from the list >*/
{
   ...
}
```

Input parameters

`l` a list (`sf_list`).

10.9 Conjugate-direction iteration for complex numbers (ccdstep.c)

10.9.1 sf_ccdstep_init

Creates a complex number list for internal storage.

Call

```
sf_ccdstep_init();
```

Definition

```
void sf_ccdstep_init(void)
/*< initialize internal storage >*/
{
   ...
}
```

10.9.2 sf_ccdstep_close

Frees the space allocated for internal storage by `sf_ccdstep_init`.

Call

```
sf_ccdstep_close();
```

Definition

```
void sf_ccdstep_close(void)
/*< free internal storage >*/
{
   ...
}
```

10.9.3 sf_ccdstep

Calculates one step for the conjugate direction iteration, that is, it calculates the new conjugate gradient for the new line search direction. It works like `sf_cdstep` but for complex numbers.

Call

```
sf_ccdstep (forget, nx, ny, x, g, rr, gg);
```

Definition

```
void sf_ccdstep(bool forget          /* restart flag */,
                int nx               /* model size */,
                int ny               /* data size */,
                sf_complex* x        /* current model [nx] */,
                const sf_complex* g  /* gradient [nx] */,
                sf_complex* rr       /* data residual [ny] */,
                const sf_complex* gg /* conjugate gradient [ny] */)
/*< Step of conjugate-direction iteration.
  The data residual is rr = A x - dat>*/
{
   ...
}
```

Input parameters

`forget`	restart flag (`bool`).
`nx`	model size (`int`).
`ny`	data size (`int`).
`x`	current model (`sf_complex*`).
`g`	gradient. Must be of type `const sf_complex*`.
`rr`	data residual (`sf_complex*`).
`gg`	conjugate gradient. Must be of type `const sf_complex*`.

10.9.4 saxpy

Multiplies a given complex number with an array of complex numbers and stores the cumulative products in another array.

Call

```
saxpy (n, a, x, y);
```

Definition

```
static void saxpy(int n, sf_double_complex a,
                  const sf_complex *x,
                  sf_complex *y)
/* y += a*x */
{
   ...
}
```

Input parameters

`n` length of the array of complex number (`int`).

`a` a complex number. Must be of type `sf_double_complex`.

`x` an array complex numbers (`sf_complex*`).

`y` location where the cumulative sum of a*x is to be stored (`sf_complex*`).

10.9.5 dsdot

Returns the Hermitian dot product of two complex numbers or the sum of the dot products if the are two arrays of complex numbers.

Call

```
prod = dsdot(n, cx, cy);
```

Definition

```
static sf_double_complex dsdot(int n,
                               const sf_complex *cx,
                               const sf_complex *cy)
/* Hermitian dot product */
{
   ...
}
```

Input parameters

`n` size of the array of complex numbers (`int`).

`cx` a complex number (`sf_complex*`).

`cy` a complex number (`sf_complex*`).

Output

`prod` dot product of the complex numbers. It is of type `static sf_double_complex`.

10.10 Linked list for conjugate-direction-type methods (complex data) (clist.c)

10.10.1 sf_clist_init

Creates an empty list for complex numbers. It returns a pointer to the list.

Call

```
sf_clist_init();
```

Definition

```
sf_clist sf_clist_init(void)
/*< create an empty list >*/
{
   ...
}
```

Output

`l`	an empty list. Must be of type `sf_clist`.

10.10.2 sf_clist_rewind

Rewinds the list, that is, it makes the pointer to the current position equal to the first entry position.

Call

```
sf_clist_rewind(l);
```

Definition

```
void sf_clist_rewind(sf_clist l)
/*< return to the start >*/
{
   ...
}
```

Input parameters

`l`	a list. Must be of type `sf_clist`.

10.10.3 sf_clist_depth

Returns the depth (length) of the list.

Input parameters

`l`	a list. Must be of type `sf_clist`.

Output

`l->depth`	depth (length) of the list (`int`).

10.10.4 sf_clist_add

Adds an entry to the list.

Call

```
sf_clist_depth(l);
```

Definition

```
int sf_clist_depth(sf_clist l)
/*< return list depth >*/
{
   ...
}
```

Call

```
sf_clist_add(l, g, gn);
```

Definition

```
void sf_clist_add(sf_clist l, sf_complex *g, double gn)
/*< add an entry in the list >*/
{
   ...
}
```

Input parameters

`l`	a list. Must be of type `sf_clist`.
`g`	value which is to be entered in the list. Must be of type exttsf_complex*.
`gn`	name or key of the value which is to be entered in the list (`double`).

10.10.5 sf_llist_down

Extracts an entry from the list.

Call

```
sf_clist_down(l, g, gn);
```

Definition

```
void sf_clist_down(sf_clist l, sf_complex **g, double *gn)
/*< extract and entry from the list >*/
{
   ...
}
```

Input parameters

`l`	a list. Must be of type `sf_clist`.
`g`	location where extracted value is to be stored (`sf_complex**`).
`gn`	location where the name or key of the value is to be stored (`double*`).

10.10.6 sf_clist_close

Frees the space allocated for the `sf_clist` object (list).

Call

```
sf_clist_close(l);
```

Definition

```
void sf_clist_close(sf_clist l)
/*< free allocated storage >*/
{
   ...
}
```

Input parameters

`l`	a list. Must be of type `sf_clist`.

10.10.7 sf_clist_chop

Removes the first entry from the list.

Call

```
sf_clist_chop(l);
```

Definition

```
void sf_clist_chop(sf_clist l)
/*< free the top entry from the list >*/
{
   ...
}
```

Input parameters

`l`	a list. Must be of type `sf_clist`.

10.11 Solving quadratic equations (quadratic.c)

Solves the equation $ax^2 + 2bx + c = 0$ and returns the smallest positive root.

10.11.1 sf_quadratic_solve

Call

```
x1 = sf_quadratic_solve (a, b, c);
```

Definition

```
float sf_quadratic_solve (float a, float b, float c)
/*< solves a x^2 + 2 b x + c == 0 for smallest positive x >*/
{
   ...
}
```

Input parameters

`a`	coefficient of x^2 (`float`).
`b`	coefficient of x (`float`).
`c`	constant term (`float`).

Output

`x1`	solution of the quadratic equation (`float`).

10.12 Zero finder (fzero.c)

10.12.1 sf_zero

Returns the zero (root) of the input function, $f(x)$ in a specified interval $[a, b]$.

Call

```
b = sf_zero ((*func)(float), a, b, fa, fb, toler, verb);
```

Definition

```
float sf_zero (float (*func)(float) /* function f(x) */,
               float a, float b     /* interval */,
               float fa, float fb   /* f(a) and f(b) */,
               float toler          /* tolerance */,
               bool verb            /* verbosity flag */)
/*< Return c such that f(c)=0 (within tolerance).
  fa and fb should have different signs. >*/
{
    float c, fc, m, s, p, q, r, e, d;
    char method[256];

   ...
    return b;
}
```

Input parameters

`(*func)(float)`	function, the root of which is required. Must be of type `sf_double_complex`.
`a`	lower limit of the interval (`float`).
`b`	upper limit of the interval (`float`).

`fa`	function value at the lower limit (`float`).
`fb`	function value at the upper limit (`float`).
`toler`	error tolerance (`float`).
`verb`	verbosity flag (`bool`).

Output

`b`	root of the input function. It is of type `float`.

10.13 Runge-Kutta ODE solvers (runge.c)

10.13.1 sf_runge_init

Initializes the required variables and allocates the required space for the ODE solver for raytracing.

Call

```
sf_runge_init(dim1, n1, d1);
```

Definition

```
void sf_runge_init(int dim1 /* dimensionality */,
                   int n1   /* number of ray tracing steps */,
                   float d1 /* step in time */)
/*< initialize >*/
{
   ...
}

\subsubsection*{Input parameters}
\begin{desclist}{\tt }{\quad}[\tt dim1]
   \setlength\itemsep{0pt}
   \item[dim1] dimension of the ODE (\texttt{int}).
   \item[n1]   number of steps for performing the raytracing (\texttt{int}).
   \item[d1]   number of time steps for performing the raytracing (\texttt{int}).
\end{desclist}



\subsection{{sf\_runge\_close\_init}}
Frees all allocated memory.

\subsubsection*{Call}
\begin{verbatim}sf_runge_close();
```

Definition

```
void sf_runge_close(void)
/*< free allocated storage >*/
{
   ...
}
```

10.13.2 sf_ode23

This function solves a first order ODE to calculate the travel time by raytracing.

Call

```
f = sf_ode23 (float t, tol, y, par,
                (*rhs)(void*,float*,float*), (*term)(void*,float*));
```

Definition

```
float sf_ode23 (float t    /* time integration */,
                float* tol /* error tolerance */,
                float* y   /* [dim] solution */,
                void* par  /* parameters for function evaluation */,
                void (*rhs)(void*,float*,float*)
                /* RHS function */,
                int (*term)(void*,float*)
             /* function returning 1 if the ray needs to terminate */)
/*< ODE solver for dy/dt = f where f comes from rhs(par,y,f)
    Note: Value of y is changed inside the function.>*/
{
   ...
}
```

Input parameters

`t`	total time for integration (`float`).
`tol`	error tolerance (`float*`).
`y`	the solution, of dimension `dim` (`float*`).
`par`	parameters to evaluate the rhs function (`void*`).
`(*rhs)(void*,float*,float*)`	function which evaluates the rhs of the ODE. Its inputs the parameters, the solution and the k's in Runge-Kutta scheme. Output is the rhs function f of the ODE (`void*`).
`(*term)(void*,float*)`	a function which returns 1 if the ray is to be terminated (`int`).

Output

`t1`	total travel time for the ray. It is of type `float`.

10.13.3 sf_ode23_step

Solves a first order ODE and returns trajectory calculated by raytracing.

Call

```
it = sf_ode23_step (y, par, (*rhs)(void*,float*,float*), (*term)(void*,float*), traj);
```

Definition

```
int sf_ode23_step (float* y    /* [dim] solution */,
                   void* par   /* parameters for function evaluation */,
                   void (*rhs)(void*,float*,float*)
                   /* RHS function */,
                   int (*term)(void*,float*)
                   /* function returning 1 if the ray needs to terminate */,
                   float** traj /* [nt+1][dim] - ray trajectory (output) */)
/*< ODE solver for dy/dt = f where f comes from rhs(par,y,f)
  Note:
  1. Value of y is changed inside the function.
  2. The output code for it = ode23_step(...)
  it=0 - ray traced to the end without termination
  it>0 - ray terminated
  The total traveltime along the ray is
  nt*dt if (it = 0); it*dt otherwise
  >*/
{
   ...
}
```

Input parameters

`y`	the solution, of dimension `dim` (`float*`).
`par`	parameters to evaluate the rhs function (`void*`).
`(*rhs)(void*,float*,float*)`	function which evaluates the rhs of the ODE. Its inputs the parameters, the solution and the k's in Runge-Kutta scheme. Output is the rhs function f of the ODE (`void*`).
`(*term)(void*,float*)`	a function which returns 1 if the ray is to be terminated (`int`).
`traj`	location where the output ray trajectory is to be stored (`float**`).

Output

`t1`	total travel time for the ray. It is of type `int`.

10.14 Solver function for iterative least-squares optimization (tinysolver.c)

10.14.1 sf_tinysolver

Performs the linear inversion for equations of the type $Lx = y$ to compute the model x.

Call

```
sf_tinysolver (Fop, stepper, nm, nd, m, m0, d, niter);
```

Definition

```
void sf_tinysolver (sf_operator Fop       /* linear operator */,
                    sf_solverstep stepper /* stepping function */,
                    int nm                /* size of model */,
                    int nd                /* size of data */,
                    float* m              /* estimated model */,
                    const float* m0       /* starting model */,
                    const float* d        /* data */,
                    int niter             /* iterations */)
/*< Generic linear solver. Solves oper{x} =~ dat >*/
{
   ...
}
```

Input parameters

`Fop`	a linear operator applied to the model x. Must be of type `sf_operator`.
`stepper`	a stepping function to perform updates on the initial model (`sf_solverstep`).
`nm`	size of the model (`int`).
`nd`	size of the data (`int`).
`m`	estimated model (`int`).
`mo`	initial model (`const float`).
`d`	data (`const float`).
`niter`	number of iterations (`int`).

10.15 Solver functions for iterative least-squares optimization (bigsolver.c)

10.15.1 sf_solver_prec

Applies solves generic linear equations after preconditioning the data.

Call

```
sf_solver_prec (oper, solv, prec, nprec, nx, ny,
                x, dat, niter, eps, "x0",x0, ..., "end");
```

Definition

```
void sf_solver_prec (sf_operator oper   /* linear operator */,
                     sf_solverstep solv /* stepping function */,
                     sf_operator prec   /* preconditioning operator */,
                     int nprec          /* size of p */,
                     int nx             /* size of x */,
                     int ny             /* size of dat */,
                     float* x           /* estimated model */,
                     const float* dat   /* data */,
                     int niter          /* number of iterations */,
                     float eps          /* regularization parameter */,
                    ...                /* variable number of arguments */)
/*< Generic preconditioned linear solver.
 ---
 Solves
 oper{x} =~ dat
 eps p   =~ 0
 where x = prec{p}
 ---
 The last parameter in the call to this function should be "end".
 Example:
 ---
 sf_solver_prec (oper_lop,sf_cgstep,prec_lop,
 np,nx,ny,x,y,100,1.0,"x0",x0,"end");
 ---
 Parameters in ...:
 ...
 "wt":     float*:         weight
 "wght":   sf_weight wght: weighting function
 "x0":     float*:         initial model
 "nloper": sf_operator:    nonlinear operator
 "mwt":    float*:         model weight
 "verb":   bool:           verbosity flag
 "known":  bool*:          known model mask
 "nmem":   int:            iteration memory
 "nfreq":  int:            periodic restart
 "xmov":   float**:        model iteration
 "rmov":   float**:        residual iteration
 "err":    float*:         final error
 "res":    float*:         final residual
 "xp":     float*:         preconditioned model
 >*/
{
   ...
}
```

Input parameters

`oper`	the operator. Must be of type `sf_operator`

`solv`	the stepping function (`sf_solverstep`).
`prec`	preconditioning operator (`sf_operator`).
`nprec`	size of the preconditioned data (`int`).
`nx`	size of the estimated model (`int`).
`ny`	size of the data (`int`).
`x`	estimated model (`float*`).
`dat`	the data (`const float*`).
`niter`	number of iterations (`int`).
`eps`	regularization parameter (`float`).
`...`	variable number of arguments.

10.15.2 sf_csolver_prec

Applies solves generic linear equations for complex data after preconditioning the data.

Call

```
sf_csolver_prec (oper, solv, prec, nprec, nx, ny,
                 x, dat, niter, eps, "x0",x0, ..., "end");
```

Definition

```
sf_csolver_prec (oper, solv, prec, nprec, nx, ny,
                 x, dat, niter, eps, niter, eps, "x0",x0, ..., "end");
```

Input parameters

`oper`	the operator (`sf_coperator`).
`solv`	the stepping function (`sf_csolverstep`).
`prec`	preconditioning operator (`sf_coperator`).
`nprec`	size of the preconditioned data (`int`).
`nx`	size of the estimated model (`int`).
`ny`	size of the data (`int`).
`x`	estimated model (`sf_complex*`).
`dat`	the data. Must be of type `const sf_complex*`.
`niter`	number of iterations (`int`).
`eps`	regularization parameter (`float`).
`...`	variable number of arguments.

10.15.3 sf_solver_reg

Applies solves generic linear equations after regularizing the data.

Call

```
sf_solver_reg (oper, solv, reg, nreg, nx, ny,
               x, y, niter, eps, "x0",x0, ..., "end");
```

Definition

```
void sf_solver_reg (sf_operator oper   /* linear operator */,
                    sf_solverstep solv /* stepping function */,
                    sf_operator reg    /* regularization operator */,
                    int nreg           /* size of reg{x} */,
                    int nx             /* size of x */,
                    int ny             /* size of dat */,
                    float* x           /* estimated model */,
                    const float* dat   /* data */,
                    int niter          /* number of iterations */,
                    float eps          /* regularization parameter */,
                   ...                /* variable number of arguments */)
/*< Generic regularized linear solver.
  ---
  Solves
  oper{x}    =~ dat
  eps reg{x} =~ 0
  ---
  The last parameter in the call to this function should be "end".
  Example:
  ---
  sf_solver_reg (oper_lop,sf_cgstep,reg_lop,
  np,nx,ny,x,y,100,1.0,"x0",x0,"end");
  ---
  Parameters in ...:

  "wt":     float*:          weight
  "wght":   sf_weight wght: weighting function
  "x0":     float*:          initial model
  "nloper": sf_operator:    nonlinear operator
  "nlreg":  sf_operator:    nonlinear regularization operator
  "verb":   bool:            verbosity flag
  "known":  bool*:           known model mask
  "nmem":   int:             iteration memory
  "nfreq":  int:             periodic restart
  "xmov":   float**:         model iteration
  "rmov":   float**:         residual iteration
  "err":    float*:          final error
  "res":    float*:          final residual
  "resm":   float*:          final model residual
  >*/
{
   ...
}
```

Input parameters

oper	the linear operator (sf_operator).
solv	the stepping function (sf_solverstep).
prec	regularization operator (sf_operator).
nreg	size of the regularized data (int).
nx	size of the estimated model (int).
ny	size of the data (int).
x	estimated model (float*).
dat	the data (const float*).
niter	number of iterations (int).
eps	regularization parameter (float).
...	variable number of arguments.

10.15.4 sf_solver

Solves generic linear equations.

Call

```
sf_solver (oper, solv, nx, ny, x, dat, niter, "x0",x0, ..., "end");
```

Definition

```
void sf_solver (sf_operator oper   /* linear operator */,
                sf_solverstep solv /* stepping function */,
                int nx             /* size of x */,
                int ny             /* size of dat */,
                float* x           /* estimated model */,
                const float* dat   /* data */,
                int niter          /* number of iterations */,
               ...                /* variable number of arguments */)
/*< Generic linear solver.
  ---
  Solves
  oper{x}    =~ dat
  ---
  The last parameter in the call to this function should be "end".
  Example:
  ---
  sf_solver (oper_lop,sf_cgstep,nx,ny,x,y,100,"x0",x0,"end");
  ---
  Parameters in ...:
  ---
  "wt":     float*:         weight
  "wght":   sf_weight wght: weighting function
  "x0":     float*:         initial model
  "nloper": sf_operator:    nonlinear operator
  "mwt":    float*:         model weight
  "verb":   bool:           verbosity flag
  "known":  bool*:          known model mask
  "nmem":   int:            iteration memory
  "nfreq":  int:            periodic restart
  "xmov":   float**:        model iteration
  "rmov":   float**:        residual iteration
  "err":    float*:         final error
  "res":    float*:         final residual
  >*/
{
   ...
}
```

Input parameters

oper	the operator (sf_operator).
solv	the stepping function (sf_solverstep).
nx	size of the estimated model (int).
ny	size of the data (int).
x	estimated model (float*).
dat	the data (const float*).
niter	number of iterations (int).
eps	regularization parameter (float).
...	variable number of arguments.

10.15.5 sf_left_solver

Solves generic linear equations for non-symmetric operators.

Call

```
sf_left_solver (oper, solv, nx, x, dat, niter, "x0",x0, ..., "end");
```

Definition

```
void sf_left_solver (sf_operator oper   /* linear operator */,
                     sf_solverstep solv /* stepping function */,
                     int nx             /* size of \texttt{x} and dat */,
                     float* x           /* estimated model */,
                     const float* dat   /* data */,
                     int niter          /* number of iterations */,
                    ...                /* variable number of arguments */)
/*< Generic linear solver for non-symmetric operators.
  ---
  Solves
  oper{x}    =~ dat
  ---
  The last parameter in the call to this function should be "end".
  Example:
  ---
  sf_left_solver (oper_lop,sf_cdstep,nx,ny,x,y,100,"x0",x0,"end");
  ---
  Parameters in ...:
  ---
  "wt":     float*:         weight
  "wght":   sf_weight wght: weighting function
  "x0":     float*:         initial model
  "nloper": sf_operator:    nonlinear operator
  "mwt":    float*:         model weight
  "verb":   bool:           verbosity flag
  "known":  bool*:          known model mask
  "nmem":   int:            iteration memory
  "nfreq":  int:            periodic restart
  "xmov":   float**:        model iteration
  "rmov":   float**:        residual iteration
  "err":    float*:         final error
  "res":    float*:         final residual
  >*/
{
   ...
}
```

Input parameters

`oper`	the operator (`sf_operator`).
`solv`	the stepping function (`sf_solverstep`).
`nx`	size of the estimated model (`int`).
`x`	estimated model (`float*`).
`dat`	the data (`const float*`).
`niter`	number of iterations (`int`).
`eps`	regularization parameter (`float`).
`...`	variable number of arguments.

10.15.6 sf_csolver

Solves generic linear equations for complex data.

Call

```
sf_csolver (oper, solv, nx, ny, x, dat, niter, "x0",x0, ..., "end");
```

Definition

```
void sf_csolver (sf_coperator oper        /* linear operator */,
                 sf_csolverstep solv      /* stepping function */,
                 int nx                   /* size of x */,
                 int ny                   /* size of dat */,
                 sf_complex* x            /* estimated model */,
                 const sf_complex* dat    /* data */,
                 int niter                /* number of iterations */,
                ...                      /* variable number of arguments */)
/*< Generic linear solver for complex data.
  ---
  Solves
  oper{x}    =~ dat
  ---
  The last parameter in the call to this function should be "end".
  Example:
  ---
  sf_csolver (oper_lop,sf_cgstep,nx,ny,x,y,100,"x0",x0,"end");
  ---
  Parameters in ...:
  ---
  "wt":     float*:          weight
  "wght":   sf_cweight wght: weighting function
  "x0":     sf_complex*:  initial model
  "nloper": sf_coperator:    nonlinear operator
  "verb":   bool:            verbosity flag
  "known":  bool*:           known model mask
  "nmem":   int:             iteration memory
  "nfreq":  int:             periodic restart
  "xmov":   sf_complex**: model iteration
  "rmov":   sf_complex**: residual iteration
  "err":    float*:  final error
  "res":    sf_complex*:  final residual
  >*/
{
   ...
}
```

Input parameters

`oper`	the operator (`sf_coperator`).
`solv`	the stepping function (`sf_csolverstep`).
`nx`	size of the estimated model (`int`).
`ny`	size of the data (`int`).
`x`	estimated model (`sf_complex*`).
`dat`	the data. Must be of type `const sf_complex*`.
`niter`	number of iterations (`int`).

`eps`	regularization parameter (`float`).
`...`	variable number of arguments.

10.16 Weighting for iteratively-reweighted least squares (irls.c)

10.16.1 sf_irls_init

Allocates the space equal to the data size for iteratively-reweighted least squares.

Call

```
sf_irls_init(n);
```

Definition

```
void sf_irls_init(int n)
/*< Initialize with data size >*/
{
   ...
}
```

Input parameters

`n`	size of the data (`int`).

10.16.2 sf_irls_close

Frees the space allocated for the iteratively-reweighted least squares by `sf_irls_init`.

Call

```
sf_irls_close();
```

Definition

```
void sf_irls_close(void)
/*< free allocated storage >*/
{
   ...
}
```

10.16.3 sf_l1

Calculates the weights for L_1 norm.

Call

```
sf_l1 (n, res, weight);
```

Definition

```
void sf_l1 (int n, const float *res, float *weight)
/*< weighting for L1 norm >*/
{
   ...
}
```

Input parameters

`n`	size of the data (`int`).
`res`	data (`const float*`).
`weight`	weights for L_1 norm (`float*`).

10.16.4 sf_cauchy

Calculates the weights for Cauchy norm.

Call

```
sf_cauchy (n, res, weight);
```

Definition

```
void sf_cauchy (int n, const float *res, float *weight)
/*< weighting for Cauchy norm >*/
{
   ...
}
```

Input parameters

`n`	size of the data (`int`).
`res`	data (`const float*`).
`weight`	weights for Cauchy norm (`float*`).

10.17 Tridiagonal matrix solver (tridiagonal.c)

10.17.1 sf_tridiagonal_init

Initializes the object of the abstract data of type `sf_tris`, which contains a matrix of size `n` with separate variables for the diagonal and off-diagonal entries and also for the solution to the matrix equation which it will be used to solve.

Call

```
slv = sf_tridiagonal_init (n);
```

Definition

```
sf_tris sf_tridiagonal_init (int n /* matrix size */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`n`	size of the matrix (`int`).

Output

`slv`	the tridiagonal solver. It is of type `sf_tris`.

10.17.2 sf_tridiagonal_define

Fills in the diagonal and off-diagonal entries in the tridiagonal solver based on the input entries `diag` and `offd`.

Call

```
sf_tridiagonal_define (slv, diag, offd);
```

Definition

```
void sf_tridiagonal_define (sf_tris slv /* solver object */,
                            float* diag /* diagonal */,
                            float* offd /* off-diagonal */)
/*< fill the matrix >*/
{
   ...
}
```

Input parameters

`slv`	the solver object (`sf_tris`).
`diag`	the diagonal (`float*`).
`offd`	the off-diagonal (`float*`).

10.17.3 sf_tridiagonal_const_define

Fills in the diagonal and off diagonal entries in the tridiagonal solver based on the input entries `diag` and `offd`. It works like `sf_tridiagonal_define` but for the special case where the matrix is Toeplitz.

Call

```
sf_tridiagonal_const_define (slv, diag, offd, damp);
```

Definition

```
void sf_tridiagonal_const_define (sf_tris slv /* solver object */,
                                  float diag  /* diagonal */,
                                  float offd  /* off-diagonal */,
                                  bool damp   /* damping */)
/*< fill the matrix for the Toeplitz case >*/
{
   ...
}
```

Input parameters

`slv`	the solver object. Must be of type `sf_tris`.
`diag`	the diagonal (`float*`).
`offd`	the off-diagonal (`float*`).
`damp`	damping (`bool`).

10.17.4 sf_tridiagonal_solve

Solves the matrix equation (like $La = b$, where b is the input for the solve function, a is the output and L is the matrix defined by `sf_tridiagonal_define`) and stores the solution in the space allocated by the variable `x` in the `slv` object.

Call

```
sf_tridiagonal_solve (sf_tris slv, b);
```

Definition

```
void sf_tridiagonal_solve (sf_tris slv /* solver object */,
                           float* b /* in - right-hand side, out - solution */)
/*< invert the matrix >*/
{
   ...
}
```

Input parameters

`slv`	the solver object. Must be of type `sf_tris`.
`b`	right hand side of the matrix equation $La = b$ (`float*`).

Definition

```
void sf_tridiagonal_solve (sf_tris slv /* solver object */,
                           float* b /* in - right-hand side, out - solution */)
/*< invert the matrix >*/
{
    int k;
    ...
}
```

10.17.5 sf_tridiagonal_close

This function frees the allocated space for the `slv` object.

Call

```
sf_tridiagonal_close (slv);
```

Definition

```
void sf_tridiagonal_close (sf_tris slv)
/*< free allocated storage >*/
{
   ...
}
```

Input parameters

`slv`	the solver object. Must be of type `sf_tris`.

Chapter 11

Interpolation

11.1 1-D interpolation (int1.c)

11.1.1 sf_int1_init

Initializes the required variables and allocates the required space for 1D interpolation.

Call

```
sf_int1_init (coord, o1, d1, n1, interp, nf_in, nd_in);
```

Definition

```
void  sf_int1_init (float* coord               /* cooordinates [nd] */,
                 float o1, float d1, int n1 /* axis */,
                 sf_interpolator interp     /* interpolation function */,
                 int nf_in                  /* interpolator length */,
                 int nd_in                  /* number of data points */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`coord`	coordinates (`float*`).
`o1`	origin of the axis (`float`).
`d1`	sampling of the axis (`float`).
`n1`	length of the axis (`float`).
`interp`	interpolation function (`sf_interpolator`).
`nf_in`	interpolator length (`int`).
`nd_in`	number of data points (`int`).

11.1.2 sf_int1_lop

Applies the linear operator for interpolation.

Call

```
sf_int1_lop (adj, add, nm, ny, x, ord);
```

Definition

```
void  sf_int1_lop (bool adj, bool add, int nm, int ny, float* x, float* ord)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether the output is `x` or `ord` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nm`	size of `x` (`int`).
`ny`	size of `ord` (`int`).
`x`	output or operator, depending on whether `adj` is true or false (`float*`).
`ord`	output or operator, depending on whether `adj` is true or false (`float*`).

11.1.3 sf_cint1_lop

Applies the complex linear operator for interpolation of complex data.

Call

```
sf_cint1_lop (adj, add, nm, ny, x, ord);
```

Definition

```
void  sf_cint1_lop (bool adj, bool add, int nm, int ny, sf_complex* x, sf_comple
x* ord)
/*< linear operator for complex numbers >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether the output is `x` or `ord` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nm`	size of `x` (`int`).
`ny`	size of `ord` (`int`).
`x`	output or operator, depending on whether `adj` is true or false (`sf_complex*`).

`ord` output or operator, depending on whether `adj` is true or false (`sf_complex*`).

11.1.4 sf_int1_close

Frees the space allocated for 1D interpolation by `sf_int1_init`.

Call

```
sf_int1_close ();
```

Definition

```
void sf_int1_close (void)
/*< free allocated storage >*/
{
   ...
}
```

11.2 2-D interpolation (int2.c)

11.2.1 sf_int2_init

Initializes the required variables and allocates the required space for 2D interpolation.

Call

```
sf_int2_init (coord, o1, o2, d1, d2, n1, n2, interp, nf_in, nd_in);
```

Definition

```
void  sf_int2_init (float** coord          /* coordinates [nd][2] */,
                    float o1, float o2,
                    float d1, float d2,
                    int   n1, int   n2     /* axes */,
                    sf_interpolator interp /* interpolation function */,
                    int nf_in              /* interpolator length */,
                    int nd_in              /* number of data points */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`coord`	coordinates (`float**`).
`o1`	origin of the first axis (`float`).
`o2`	origin of the second axis (`float`).
`d1`	sampling of the first axis (`float`).
`d2`	sampling of the second axis (`float`).

`n1`	length of the first axis (`float`).
`n2`	length of the second axis (`float`).
`interp`	interpolation function (`sf_interpolator`).
`nf_in`	interpolator length (`int`).
`nd_in`	number of data points (`int`).

11.2.2 sf_int2_lop

Applies the linear operator for 2D interpolation.

Call

```
sf_int2_lop (adj, add, nm, ny, x, ord);
```

Definition

```
void  sf_int2_lop (bool adj, bool add, int nm, int ny, float* x, float* ord)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether the output is `x` or `ord` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nm`	size of `x` (`int`).
`ny`	size of `ord` (`int`).
`x`	output or operator, depending on whether `adj` is true or false (`float*`).
`ord`	output or operator, depending on whether `adj` is true or false (`float*`).

11.2.3 sf_int2_close

Frees the space allocated for 2D interpolation by `sf_int2_init`.

Call

```
sf_int2_close();
```

Definition

```
void sf_int2_close (void)
/*< free allocated storage >*/
{
   ...
}
```

11.3 3-D interpolation (int3.c)

11.3.1 sf_int3_init

Initializes the required variables and allocates the required space for 3D interpolation.

Call

```
sf_int3_init (coord, o1,o2,o3, d1,d2,d3,
              n1,n2,n3, interp, nf_in, nd_in);
```

Definition

```
void  sf_int3_init (float** coord          /* coordinates [nd][3] */,
                    float o1, float o2, float o3,
                    float d1, float d2, float d3,
                    int   n1, int   n2,   int n3 /* axes */,
                    sf_interpolator interp /* interpolation function */,
                    int nf_in              /* interpolator length */,
                    int nd_in              /* number of data points */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`coord`	coordinates (`float**`).
`o1`	origin of the first axis (`float`).
`o2`	origin of the second axis (`float`).
`o3`	origin of the third axis (`float`).
`d1`	sampling of the first axis (`float`).
`d2`	sampling of the second axis (`float`).
`d3`	sampling of the third axis (`float`).
`n1`	length of the first axis (`float`).
`n2`	length of the second axis (`float`).
`n3`	length of the third axis (`float`).
`interp`	interpolation function (`sf_interpolator`).
`nf_in`	interpolator length (`int`).
`nd_in`	number of data points (`int`).

11.3.2 sf_int3_lop

Applies the linear operator for 3D interpolation.

Call

```
sf_int3_lop (adj, add, nm, ny, mm, dd);
```

Definition

```
void  sf_int3_lop (bool adj, bool add, int nm, int ny, float* mm, float* dd)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether the output is `x` or `ord` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nm`	size of `x` (`int`).
`ny`	size of `ord` (`int`).
`x`	output or operator, depending on whether `adj` is true or false (`float*`).
`ord`	output or operator, depending on whether `adj` is true or false (`float*`).

11.3.3 sf_int3_close

Frees the space allocated for 3D interpolation by `sf_int3_init`.

Call

```
int3_close ();
```

Definition

```
void int3_close (void)
/*< free allocated storage >*/
{
   ...
}
```

11.4 Basic interpolation functions (interp.c)

11.4.1 sf_bin_int

Computes the nearest neighbor interpolation function coefficients.

Call

```
sf_bin_int (x, n, w);
```

Definition

```
void sf_bin_int (float x, int n, float* w)
/*< nearest neighbor >*/
{
   ...
}
```

Input parameters

`x` data (`float`).

`n` number of interpolation points (`int`).

`w` interpolation coefficients (`float*`).

11.4.2 sf_lin_int

Computes the linear interpolation function coefficients.

Call

```
sf_lin_int (x, n, w);
```

Definition

```
void sf_lin_int (float x, int n, float* w)
/*< linear >*/
{
   ...
}
```

Input parameters

`x` data (`float`).

`n` number of interpolation points (`int`).

`w` interpolation coefficients (`float*`).

11.4.3 sf_lg_int

Computes the Lagrangian interpolation function coefficients.

Call

```
sf_lg_int (x, n, w);
```

Definition

```
void sf_lg_int (float x, int n, float* w)
/*< Lagrangian >*/
{
   ...
}
```

Input parameters

`x`	data (`float`).
`n`	number of interpolation points (`int`).
`w`	interpolation coefficients (`float*`).

11.4.4 sf_taylor_int

Computes the taylor interpolation function coefficients.

Call

```
sf_taylor (x, n, w);
```

Definition

```
void sf_taylor (float x, int n, float* w)
/*< Taylor >*/
{
   ...
}
```

Input parameters

`x`	data (`float`).
`n`	number of interpolation points (`int`).
`w`	interpolation coefficients (`float*`).

11.5 Convert data to B-spline coefficients by fast B-spline transform (prefilter.c)

11.5.1 sf_prefilter_init

Initializes the pre-filter for spline interpolation by initializing the required variables and allocating the required space.

Call

```
sf_prefilter_init (nw, nt_in, pad_in);
```

Definition

```
void sf_prefilter_init (int nw     /* spline order */,
                        int nt_in  /* temporary storage length */,
                        int pad_in /* padding */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`nw`	order of the spline (`int`).
`nt_in`	length of the temporary storage (`int`).
`pad_in`	length of the padding required (`int`).

11.5.2 sf_prefilter_apply

Applies the pre-filter to the input 1D data to convert it to the spline coefficients.

Call

```
sf_prefilter_apply (nd, dat);
```

Definition

```
void sf_prefilter_apply (int nd     /* data length */,
                         float* dat /* in - data, out - coefficients */)
/*< Convert 1-D data to spline coefficients >*/
{
   ...
}
```

Input parameters

`nd`	length of the input data (`int`).
`dat`	the input data, which is converted to spline coefficients as output (`float*`).

11.5.3 sf_prefilter

Applies the pre-filter to the input N dimensional data to convert it to the spline coefficients.

Call

```
sf_prefilter (dim, n, dat);
```

Definition

```
void sf_prefilter (int dim    /* number of dimensions */,
                   int* n     /* data size [dim] */,
                   float* dat /* in - data, out - coefficients */)
/*< Convert N-D data to spline coefficients >*/
{
   ...
}
```

Input parameters

`dim`	number of dimensions in the input data (`int`).
`n`	size of the input data (`int*`).
`dat`	the input data, which is converted to spline coefficients as output (`float*`).

11.5.4 sf_prefilter_close

Frees the space allocated for the pre-filer by `sf_prefilter_init`.

Call

```
sf_prefilter_close();
```

Definition

```
void sf_prefilter_close( void)
/*< free allocated storage >*/
{
   ...
}
```

11.6 B-spline interpolation (spline.c)

11.6.1 sf_spline_init

Initializes and defines a banded matrix for spline interpolation.

Call

```
slv = sf_spline_init (nw, nd);
```

Definition

```
sf_bands sf_spline_init (int nw /* interpolator length */,
                         int nd /* data length */)
/*< initialize a banded matrix >*/
{
   ...
}
```

Input parameters

`nw`	length of the interpolator (`int`).
`nd`	length of the data (`int`).

Output

`slv`	an object of type `sf_band`. It is of type `sf_band`.

11.6.2 sf_spline4_init

Initializes and defines a tridiagonal matrix for cubic spline interpolation.

Call

```
slv = sf_spline4_init(nd);
```

Definition

```
sf_tris sf_spline4_init (int nd /* data length */)
/*< initialize a tridiagonal matrix for cubic splines >*/
{
   ...
}
```

Input parameters

`nd`	length of the data (`int`).

Output

`slv`	an object of type `sf_tri`. It is of type `sf_tri`.

11.6.3 sf_spline4_post

Performs the cubic spline post filtering.

Call

```
sf_spline4_post (n, n1, n2, inp, out);
```

Definition

```
void sf_spline4_post (int n            /* total trace length */,
                      int n1           /* start point */,
                      int n2           /* end point */,
                      const float* inp /* spline coefficients */,
                      float* out       /* function values */)
/*< cubic spline post-filtering >*/
{
   ...
}
```

Input parameters

`n`	total length of the trace (`int`).
`n1`	start point (`int`).
`n2`	end point (`int`).
`inp`	spline coefficients (`const float*`).
`out`	function values (`float*`).

11.6.4 sf_spline_post

Performs the post filtering to convert spline coefficients to model.

Call

```
sf_spline_post(nw, o, d, n, modl, datr);
```

Definition

```
void sf_spline_post (int nw, int o, int d, int n,
                     const float *modl, float *datr)
/*< post-filtering to convert spline coefficients to model >*/
{
   ...
}
```

Input parameters

`nw`	length of the interpolator (`int`).
`o`	start point (`int`).
`d`	step size (`int`).
`n`	total length of the trace (`int`).
`modl`	spline coefficients, which have to be converted to model coefficients. Must be of type `const float*`.
`datr`	model, it is the output (`float*`).

11.6.5 sf_spline2

Performs pre-filtering for spline interpolation for 2D data.

call

```
sf_spline2 (slv1, slv2, n1, n2, dat, tmp);
```

Definition

```
void sf_spline2 (sf_bands slv1, sf_bands slv2,
                 int n1, int n2, float** dat, float* tmp)
/*< 2-D spline pre-filtering >*/
{
   ...
}
```

Input parameters

`slv1`	first banded matrix. Must be of type `sf_band`.
`slv2`	second banded matrix. Must be of type `sf_band`.
`n1`	data length on the first axis (`int`).
`n2`	data length on the second axis (`int`).
`dat`	2D data. Must be of type `const float*`*.
`tmp`	temporary arrays for calculation (`float*`).

11.7 Inverse linear interpolation (stretch.c)

11.7.1 sf_stretch_init

Initializes the object of the abstract data of type `sf_map`, which will be used to define and transform (stretch) coordinates.

Call

```
sf_map sf_stretch_init (n1, o1, d1, nd, eps, narrow);
```

Definition

```
sf_map sf_stretch_init (int n1, float o1, float d1 /* regular axis */,
                        int nd                     /* data length */,
                        float eps                  /* regularization */,
                        bool narrow                /* if zero boundary */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`n1`	axis (`int`).
`o1`	first sample on the axis (`int`).
`d1`	step length to access the sample on the same axis (`int`).
`eps`	regularizaton (`float`).
`narrow`	is boundary value zero or not (`bool`).

Output

`str`	the `sf_map` object. It is of type `sf_map`.

11.7.2 sf_stretch_define

Defines the coordinates for mapping (which in this case is stretching. That is, it fills the required variables in the `sf_map` object to map the input coordinates.

Call

```
sf_stretch_define (str, coord);
```

Definition

```
void sf_stretch_define (sf_map str, const float* coord)
/*< define coordinates >*/
{
   ...
}
```

Input parameters

`str`	the `sf_map` object. Must be of type `sf_map`.
`coor`	input coordinates (`const float`).

11.7.3 sf_stretch_apply

Converts the ordinates (`ord`) defined in the input to model (`mod`). It uses the `sf_tridiagonal_solve` function.

Call

```
sf_stretch_apply (str, ord, mod);
```

Definition

```
void sf_stretch_apply (sf_map str, const float* ord, float* mod)
/*< convert ordinates to model >*/
{
   ...
}
```

Input parameters

`str`	the `sf_map` object. Must be of type `sf_map`.
`ord`	input ordinates (`const float*`).
`mod`	model (`const float*`).

11.7.4 sf_stretch_invert

Converts model (mod) to ordinates by linear interpolation. It is the inverse of `sf_stretch_apply`.

Call

```
sf_stretch_invert (str, ord, mod);
```

Definition

```
void sf_stretch_invert (sf_map str, float* ord, const float* mod)
/*< convert model to ordinates by linear interpolation >*/
{
   ...
}
```

Input parameters

`str`	the `sf_map` object. Must be of type `sf_map`.
`ord`	input ordinates (`const float*`).
`mod`	model (`const float*`).

11.7.5 sf_stretch_close

This function frees the allocated space for the `sf_map` object.

Call

```
sf_stretch_close (str);
```

Definition

```
void sf_stretch_close (sf_map str)
/*< free allocated storage >*/
{
   ...
}
```

Input parameters

`str`	the `sf_map` object. Must be of type `sf_map`.

11.8 1-D ENO interpolation (eno.c)

11.8.1 sf_eno_init

Initializes an object of type `sf_eno` for interpolation.

Call

```
ent = sf_eno_init (order, n);
```

Definition

```
sf_eno sf_eno_init (int order /* interpolation order */,
              int n     /* data size */)
/*< Initialize interpolation object. >*/
{
   ...
}
```

Input parameters

`order`	order of interpolation (`int`).
`n`	size of the data (`int`).

Output

`ent`	an object for interpolation. It is of type `sf_eno`.

11.8.2 sf_eno_close

Frees the space allocated for the internal storage by `sf_eno_init`.

Call

```
sf_eno_close (ent);
```

Definition

```
void sf_eno_close (sf_eno ent)
/*< Free internal storage >*/
{
   ...
}
```

11.8.3 sf_eno_set

Creates a table for interpolation.

Call

```
sf_eno_set (ent, c);
```

Definition

```
void sf_eno_set (sf_eno ent, float* c /* data [n] */)
/*< Set the interpolation table. c can be changed or freed afterwords >*/
{
   ...
}
```

Input parameters

`ent`	an object for interpolation. It is of type `sf_eno`.
`c`	the data which is to be interpolated (`float*`).

11.8.4 sf_eno_apply

Interpolates the data.

Call

```
sf_eno_apply (ent, i, x, f, f1, what);
```

Definition

```
void sf_eno_apply (sf_eno ent,
                int i     /* grid location */,
                float x   /* offset from grid */,
                float *f  /* output data value */,
                float *f1 /* output derivative */,
                der what  /* flag of what to compute */)
/*< Apply interpolation >*/
{
   ...
}
```

Input parameters

`ent`	an object for interpolation. It is of type `sf_eno`.
`i`	location of the grid (`int`).
`x`	offset from the grid (`float`).
`f`	output data value (`float*`).
`f1`	output derivative (`float*`).
`what`	whether the function value or the derivative is required. Must be of type `der`.

11.9 ENO interpolation in 2-D (eno2.c)

11.9.1 sf_eno2_init

Initializes an object of type `sf_eno`2 for interpolation of 2D data.

Call

```
pnt =  sf_eno2_init (order, n1, n2);
```

Definition

```
sf_eno2 sf_eno2_init (int order      /* interpolation order */,
                      int n1, int n2 /* data dimensions */)
/*< Initialize interpolation object >*/
{
    sf_eno2 pnt;
    int i2;

   ...
    return pnt;
}
```

Input parameters

`order`	interpolation order (`int`).
`n1`	first dimension of the data (`int`).
`n2`	second dimension of the data (`int`).

Output

`pnt`	object for interpolation. It is of type `sf_eno`2.

11.9.2 sf_eno2_set

Sets the interpolation table for the 2D data in a 2D array.

Call

```
sf_eno2_set (pnt, c);
```

Definition

```
void sf_eno2_set (sf_eno2 pnt, float** c /* data [n2][n1] */)
/*< Set the interpolation table. c can be changed or freed afterwords. >*/
{
   ...
}
```

Input parameters

`pnt`	object for interpolation. It is of type `sf_eno2`.
`c`	the data (`float**`).

11.9.3 sf_eno2_set1

Sets the interpolation table for the 2D data in a 1D array, which is of size `n1*n2`.

Call

```
sf_eno2_set1 (pnt, c);
```

Definition

```
void sf_eno2_set1 (sf_eno2 pnt, float* c /* data [n2*n1] */)
/*< Set the interpolation table. c can be changed or freed afterwords. >*/
{
   ...
}
```

Input parameters

`pnt`	object for interpolation. It is of type `sf_eno2`.
`c`	the data (`float*`).

11.9.4 sf_eno2_close

Frees the space allocated for the internal storage by `sf_eno2_init`.

Call

```
sf_eno2_close (pnt);
```

Definition

```
void sf_eno2_close (sf_eno2 pnt)
/*< Free internal storage >*/
{
   ...
}
```

11.9.5 sf_eno2_apply

Interpolates the 2D data.

Call

```
sf_eno2_apply (pnt, i, j, x, y, f, f1, what);
```

Definition

```
void sf_eno2_apply (sf_eno2 pnt,
                    int i, int j     /* grid location */,
                    float x, float y /* offset from grid */,
                    float* f         /* output data value */,
                    float* f1        /* output derivative [2] */,
                    der what         /* what to compute [FUNC,DER,BOTH] */)
/*< Apply interpolation. >*/
{
   ...
}
```

Input parameters

`pnt`	an object for interpolation. It is of type `sf_eno2`.
`i`	location of the grid for first dimension (`int`).
`j`	location of the grid for second dimension (`int`).
`x`	offset from the grid for the first dimension (`float`).
`y`	offset from the grid for the second dimension (`float`).
`f`	output data value (`float*`).
`f1`	output derivative (`float*`).
`what`	whether the function value or the derivative or both are required. Must be of type `der`.

11.10 1-D ENO power-p interpolation (pweno.c)

11.10.1 sf_pweno_init

Initializes an object of type `sf_pweno`.

Call

```
ent = sf_pweno sf_pweno_init (int order, n);
```

Definition

```
sf_pweno sf_pweno_init (int order /* interpolation order */,
                        int n     /* data size */)
/*< Initialize interpolation object >*/
{
   ...
}
```

Input parameters

`order`	order of interpolation (`int`).
`n`	size of the data (`int`).

Output

`ent` object of type `sf_pweno`.

11.10.2 sf_pweno_close

Frees the space allocated for the `sf_pweno` object by `sf_pweno_init`.

Call

```
sf_pweno_close (ent);
```

Definition

```
void sf_pweno_close (sf_pweno ent)
/*< Free internal storage >*/
{
   ...
}
```

Input parameters

`ent` object of type `sf_pweno`.

11.10.3 powerpeno

Calculates the Power-p limiter for eno method using the input numbers `x` and `y`.

Call

```
power = powerpeno (x, y, p);
```

Definition

```
float powerpeno (float x, float y, int p /* power order */)
/*< Limiter power-p eno >*/
{
   ...
}
```

Input parameters

`x`	an input number (`float`).
`y`	an input number (`float`).
`p`	power order (`int`).

Output

`mins * power`	limiter power-p. It is of type `float`.

11.10.4 sf_pweno_set

Sets the interpolation undivided difference table.

Call

```
void sf_pweno_set (sf_pweno ent, float* c /* data [n] */, int p);
```

Definition

```
void sf_pweno_set (sf_pweno ent, float* c /* data [n] */, int p /* power order */)
/*< Set the interpolation undivided difference table. c can be changed or freed
afterwards >*/
{
   ...
}
```

Input parameters

`ent`	the interpolation object. Must be of type `sf_pweno`.
`c`	input data (`float*`).
`p`	power order (`int`).

11.10.5 sf_pweno_apply

Applies the interpolation.

Call

```
sf_pweno_apply (ent, i, x, f, f1, what);
```

Definition

```
void sf_pweno_apply (sf_pweno ent,
                int i     /* grid location */,
                float x   /* offset from grid */,
                float *f  /* output data value */,
                float *f1 /* output derivative */,
                derr what /* flag of what to compute */)
/*< Apply interpolation >*/
{
   ...
}
```

Input parameters

`i`	location of the grid (`int`).
`x`	offset from the grid (`float`).
`f`	output data value (`float*`).
`f1`	output derivative (`float*`).
`what`	flag of what to compute. Must be of type `derr`.

Chapter 12

Smoothing

12.1 1-D triangle smoothing as a linear operator (triangle1.c)

12.1.1 sf_triangle1_init

Initializes the triangle filter.

Call

```
sf_triangle1_init (nbox, ndat);
```

Definition

```
void sf_triangle1_init (int nbox /* triangle size */,
                        int ndat /* data size */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`inbox`	size of the triangle filter (`int`).
`ndat`	size of the data (`int`).

12.1.2 sf_triangle1_lop

Applies the triangle smoothing to one of the input data and applies the smoothed data to the unsmoothed one as a linear operator.

Call

```
sf_triangle1_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_triangle1_lop (bool adj, bool add, int nx, int ny, float* x, float* y)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether weights are applied to `yy` or `xx` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `x` (`int`).
`ny`	size of `y` (`int`).
`x`	data or operator, depending on whether `adj` is true or false (`float`).
`y`	data or operator, depending on whether `adj` is true or false. Must be of type `float`.

Output

`x or y`	the output depending on whether `adj` is true or false (`float`).

12.1.3 sf_triangle1_close

Frees the space allocated for the triangle smoothing filter.

Call

```
sf_triangle1_close();
```

Definition

```
void sf_triangle1_close(void)
/*< free allocated storage >*/
{
   ...
}
```

12.2 2-D triangle smoothing as a linear operator (triangle2.c)

12.2.1 sf_triangle2_init

Initializes the triangle filter.

Call

```
sf_triangle2_init (nbox1,nbox2, ndat1,ndat2, nrep);
```

Definition

```
void sf_triangle2_init (int nbox1, int nbox2 /* triangle size */,
                        int ndat1, int ndat2 /* data size */,
                        int nrep /* repeat smoothing */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`inbox1`	size of the triangle filter (`int`).
`inbox2`	size of the second triangle filter (`int`).
`ndat1`	size of the data (`int`).
`ndat2`	size of the second data set (`int`).
`nrep`	number of times the smoothing is to be repeated (`int`).

12.2.2 sf_triangle2_lop

Applies the triangle smoothing to one of the input data and applies the smoothed data to the unsmoothed one as a linear operator. This is just like `sf_triangle1_lop` but with two triangle filters instead of one.

Call

```
sf_triangle2_lop (adj, add, nx, ny, x, y);
```

Definition

```
void sf_triangle2_lop (bool adj, bool add, int nx, int ny, float* x, float* y)
/*< linear operator >*/
{
   ...
}
```

Input parameters

`adj`	a parameter to determine whether weights are applied to `yy` or `xx` (`bool`).
`add`	a parameter to determine whether the input needs to be zeroed (`bool`).
`nx`	size of `x` (`int`).
`ny`	size of `y` (`int`).
`x`	data or operator, depending on whether `adj` is true or false (`float`).
`y`	data or operator, depending on whether `adj` is true or false (`float`).

Output

`x or y`	the output depending on whether `adj` is true or false (`float`).

12.2.3 sf_triangle2_close

Frees the space allocated for the triangle smoothing filters.

Call

```
sf_triangle2_close();
```

Definition

```
void sf_triangle2_close(void)
/*< free allocated storage >*/
{
   ...
}
```

12.3 Triangle smoothing (triangle.c)

12.3.1 sf_triangle_init

Initializes the triangle smoothing filter.

Call

```
tr = sf_triangle_init (nbox, ndat);
```

Definition

```
sf_triangle sf_triangle_init (int nbox /* triangle length */,
                              int ndat /* data length */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`nbox`	an integer which specifies the length of the filter (`int`).
`ndat`	an integer which specifies the length of the data (`int`).

Output

`tr`	the triangle smoothing filter. It is of type `sf_triangle`.

12.3.2 fold

Folds the edges of the smoothed data, because when the data is convolved with the data, the length of the data increases and in most cases it is required that the smoothed data is of the same length as the input data.

Call

```
fold (o, d, nx, nb, np, x, tmp);
```

Definition

```
static void fold (int o, int d, int nx, int nb, int np,
                  const float *x, float* tmp)
{
   ...
}
```

Input parameters

`o`	first indices of the input data (`int`).
`d`	step size (`int`).
`nx`	data length (`int`).
`nb`	filter length (`int`).
`np`	length of the tmp array in the `sf_Triangle` data structure (`int`).
`x`	a pointer to the input data (`const float`).
`tmp`	a pointer to an array in the `sf_Triangle` data structure. Must be of type `float`

12.3.3 fold2

Is the same as `fold` except for the fact that it copies from tmp to data, unlike the fold which does it the other way round.

Call

```
fold2 (o, d, nx, nb, np, x, tmp);
```

Definition

```
static void fold2 (int o, int d, int nx, int nb, int np,
                  float *x, const float* tmp)
{
   ...
}
```

Input parameters

`o`	first indices of the input data (`int`).
`d`	step size (`int`).
`nx`	data length (`int`).
`nb`	filter length (`int`).
`np`	length of the tmp array in the `sf_Triangle` data structure. Must be of type `int`
`x`	a pointer to the input data (`const float`).
`tmp`	a pointer to an array in the `sf_Triangle` data structure (`float`).

12.3.4 doubint

Integrates the input data first in the backward direction and then if the input variable `der` is true it integrates the result forward.

Call

```
doubint (nx, xx, der);
```

Definition

```
static void doubint (int nx, float *xx, bool der)
{
   ...
}
```

Input parameters

`nx`	data length (`int`).
`xx`	a pointer to the input data (`const float`).
`der`	a parameter to specify whether forward integration is required or not (`const float`).

12.3.5 doubint2

Unlike the `doubint` function this function integrates the input data first in the forward direction and then if the input variable `der` is true it integrates the result backward direction.

Call

```
doubint2 (nx, xx, der);
```

Definition

```
static void doubint2 (int nx, float *xx, bool der)
{
   ...
}
```

Input parameters

`nx`	data length (`int`).
`xx`	a pointer to the input data (`const float`).
`der`	a parameter to specify whether forward integration is required or not (`const float`).

12.3.6 triple

Does the smoothing to the input data.

Call

```
triple (o, d, nx, nb, x, tmp, box);
```

Definition

```
static void triple (int o, int d, int nx, int nb, float* x,
                    const float* tmp, bool box)
{
    ...
}
```

Input parameters

`o`	first indices of the input data (`int`).
`d`	step size (`int`).
`nx`	data length (`int`).
`nb`	filter length (`int`).
`np`	length of the tmp array in the sf_Triangle data structure. Must be of type `int`
`x`	a pointer to the input data (`const float`).
`tmp`	a pointer to an array in the `sf_Triangle` data structure (`float`).
`box`	a parameter to specify whether a box filter is required (`bool`).

12.3.7 triple2

Does the smoothing to the input data.

Call

```
triple2 (o, d, nx, nb, x, tmp, box);
```

Definition

```
static void triple2 (int o, int d, int nx, int nb,
                    const float* x, float* tmp, bool box)
{
    ...
}
```

Input parameters

`o`	first indices of the input data (`int`).
`d`	step size (`int`).
`nx`	data length (`int`).
`nb`	filter length (`int`).
`np`	length of the tmp array in the `sf_Triangle` data structure. Must be of type `int`
`x`	a pointer to the input data (`const float`).
`tmp`	a pointer to an array in the `sf_Triangle` data structure (`float`).
`box`	a parameter to specify whether a box filter is required (`bool`).

12.3.8 sf_smooth

Smoothes the input data by first applying the `fold` function then `doubint` and then `triple`.

Call

```
sf_smooth (tr, o, d, der, box, x);
```

Definition

```
void sf_smooth (sf_triangle tr  /* smoothing object */,
                int o, int d    /* trace sampling */,
                bool der        /* if derivative */,
                bool box        /* if box filter */,
                float *x        /* data (smoothed in place) */)
/*< apply triangle smoothing >*/
{
   ...
}
```

Input parameters

`tr`	an object (filter) used for smoothing, box or triangle. Must be of type `sf_triangle`.
`o`	first indices of the input data (`int`).
`d`	step size (`int`).
`der`	a parameter to specify whether forward integration in `doubint` is required or not (`const float`).
`x`	a pointer to the input data (`float`).
`box`	a parameter to specify whether a box filter is required (`bool`).

12.3.9 sf_smooth2

Smoothes the input data by first applying the `triple2` function then `doubint2` and then `fold2`.

Call

```
sf_smooth2 (tr, o, d, der, box, x);
```

Definition

```
void sf_smooth2 (sf_triangle tr  /* smoothing object */,
                int o, int d    /* trace sampling */,
                bool der        /* if derivative */,
                bool box        /* if box filter */,
                float *x        /* data (smoothed in place) */)
/*< apply adjoint triangle smoothing >*/
{
   ...
}
```

Input parameters

`tr`	an object (filter) used for smoothing, box or triangle. Must be of type `sf_triangle`.
`o`	first indices of the input data (`int`).
`d`	step size (`int`).
`der`	a parameter to specify whether forward integration in `doubint` is required or not (`const float`).
`x`	a pointer to the input data (`float`).
`box`	a parameter to specify whether a box filter is required (`bool`).

12.3.10 sf_triangle_close

Frees the space allocated for the triangle smoothing filter.

Call

```
sf_triangle_close(tr);
```

Definition

```
void  sf_triangle_close(sf_triangle tr)
/*< free allocated storage >*/
{
   ...
}
```

Input parameters

`tr`	the triangle smoothing filter. Must be of type `sf_triangle`.

12.4 Smooth gradient operations (edge.c)

12.4.1 sf_grad2

Calculates the gradient squared of the input with the centered finite-difference formula.

Call

```
sf_grad2 (n, x, w);
```

Definition

```
void sf_grad2 (int n          /* data size */,
               const float *x /* input trace [n] */,
               float *w       /* output gradient squared [n] */)
/*< centered finite-difference gradient >*/
{
   ...
}
```

Input parameters

`n`	size of the data (`int`).
`x`	input trace (`const float*`).
`w`	output gradient squared (`float*`).

12.4.2 sf_sobel

Calculates the 9-point Sobel's gradient for a 2D image.

Call

```
sf_sobel (n1, n2, x, w1, w2);
```

Definition

```
void sf_sobel (int n1, int n2         /* data size */,
               float **x              /* input data [n2][n1] */,
               float **w1, float **w2 /* output gradient [n2][n1] */)
/*< Sobel's 9-point gradient >*/
{
   ...
}
```

Input parameters

`n1`	size of the data, first axis (`int`).
`n2`	size of the data, second axis (`int`).
`x`	2D input data (`const float**`).
`w1`	output gradient, first axis (`float**`).
`w2`	output gradient, second axis (`float**`).

12.4.3 sf_sobel2

Calculates the Sobel's gradient squared for a 2D image. It works like `sf_sobel` but outputs the gradient squared.

Call

```
sf_sobel2 (n1, n2, x, w);
```

Definition

```
void sf_sobel2 (int n1, int n2  /* data size */,
                float **x       /* input data [n2][n1] */,
                float **w       /* output gradient squared [n2][n1] */)
/*< Sobel's gradient squared >*/
{
   ...
}
```

Input parameters

n1 size of the data, first axis (int).

n2 size of the data, second axis (int).

x 2D input data (const float**).

w output gradient squared (float**).

12.4.4 sf_sobel32

Calculates the Sobel's gradient squared for a 3D image. It works like sf_sobel but outputs the gradient squared for 3D data.

Call

```
sf_sobel32 (n1, n2, n3, x, w);
```

Definition

```
void sf_sobel32 (int n1, int n2, int n3  /* data size */,
                 float ***x              /* input data [n3][n2][n1] */,
                 float ***w              /* output gradient squared */)
/*< Sobel's gradient squared in 3-D>*/
{
   ...
}
```

Input parameters

n1 size of the data, first axis (int).

n2 size of the data, second axis (int).

n3 size of the data, third axis (int).

x 3D input data (const float***).

w output gradient squared (float***).

Chapter 13

Ray tracing

13.1 Cell ray tracing (celltrace.c)

13.1.1 sf_celltrace_init

Initializes the object `sf_celltrace` for ray tracing by initializing the required variables and allocating the required space.

Call

```
ct = sf_celltrace_init (order, nt, nz, nx, dz, dx, z0, x0, slow);
```

Definition

```
sf_celltrace sf_celltrace_init (int order   /* interpolation accuracy */,
                                int nt      /* maximum time steps */,
                                int nz      /* depth samples */,
                                int nx      /* lateral samples */,
                                float dz    /* depth sampling */,
                                float dx    /* lateral sampling */,
                                float z0    /* depth origin */,
                                float x0    /* lateral origin */,
                                float* slow /* slowness [nz*nx] */)
/*< Initialize ray tracing object >*/
{
  ...
}
```

Input parameters

`order`	accuracy of the interpolation (`int`).
`nt`	maximum number of time steps (`int`).
`nz`	number of depth samples (`int`).
`nx`	number of lateral samples (`int`).
`dz`	depth sampling interval (`float`).

`dx`	lateral sampling interval (`float`).
`z0`	depth origin (`float`).
`x0`	lateral origin (`float`).
`slow`	slowness (`float*`).

Output

`ct`	the ray tracing object. It is of type `sf_celltrace`.

13.1.2 sf_celltrace_close

Frees the space allocated for the `sf_celltrace` object by `sf_celltrace_init`.

Call

```
sf_celltrace_close (ct);
```

Definition

```
void sf_celltrace_close (sf_celltrace ct)
/*< Free allocated storage >*/
{
   ...
}
```

13.1.3 sf_cell_trace

Traces the ray with the ray parameter specified in the input.

Call

```
t = sf_cell_trace (ct, xp, p, it, traj);
```

Definition

```
float sf_cell_trace (sf_celltrace ct,
                     float* xp    /* position */,
                     float* p     /* ray parameter */,
                     int* it      /* steps till boundary */,
                     float** traj /* trajectory */)
/*< ray trace >*/
{
   ...
}
```

Input parameters

`ct` the ray tracing object. It is of type `sf_celltrace`.

`xp` position (`float*`).

`p` ray parameters (`float*`).

`it` number steps till the boundary (`int*`).

`traj` trajectory of the ray (`float**`).

Output

`t` the travel time obtained by the ray tracing. It is of type `float`.

13.2 Cell ray tracing (cell.c)

13.2.1 sf_cell1_intersect

Intersects a straight ray with the cell boundary.

Call

```
sf_cell1_intersect (a, x, dy, p, sx, jx);
```

Definition

```
void sf_cell1_intersect (float a, float x, float dy, float p,
                         float *sx, int *jx)
/*< intersecting a straight ray with cell boundaries >*/
{
   ...
}
```

Input parameters

`a` gradient of slowness (`float`).

`x` non-integer part of the position in the grid relative to grid origin. It is of type `float`.

`dy` depth or lateral sampling divided by slowness. It is of type `float`.

`p` the ray parameter. It is of type `float`.

`sx` distance traveled in the medium (cell) times the velocity of the medium (equivalent to the optical path length in optics). It is of type `float*`.

`jx` the direction of the ray. It is of type `int*`.

13.2.2 sf_cell1_update1

Performs the first step of the first order symplectic method for ray tracing.

Call

```
tt = sf_cell1_update1 (dim, s, v, p, g);
```

Definition

```
float sf_cell1_update1 (int dim, float s, float v, float *p, const float *g)
/*< symplectic first-order: step 1 >*/
{
   ...
}
```

Input parameters

`dim`	dimension (`int`).
`s`	σ (`float`).
`v`	slowness. It is of type `float`.
`p`	direction. It is of type `float*`.
`g`	slowness gradient. It is of type `const float*`.

Output

0.5*v*v*s*(1. + s*pg)	travel time. It is of type `float`.

13.2.3 sf_cell1_update2

Performs the second step of the first order symplectic method for ray tracing.

Call

```
tt = sf_cell1_update2 (dim, s, v, p, g);
```

Definition

```
float sf_cell1_update2 (int dim, float s, float v, float *p, const float *g)
/*< symplectic first-order: step 2 >*/
{
   ...
}
```

Input parameters

`dim`	dimension (`int`).
`s`	σ (`float`).
`v`	slowness. It is of type `float`.
`p`	direction. It is of type `float*`.
`g`	slowness gradient. It is of type `const float*`.

Output

0.5*v*v*s*(1. - s*pg)	travel time. It is of type `float`.

13.2.4 sf_cell11_intersect2

Intersects a straight ray with the cell boundary.

Call

```
sf_cell11_intersect2 (a, da, p, g, sp, jp);
```

Definition

```
void sf_cell11_intersect2 (float a, float da,
                           const float* p, const float* g,
                           float *sp, int *jp)
/*< intersecting a straight ray with cell boundaries >*/
{
   ...
}
```

Input parameters

a   position in the grid (`float`).

da   grid spacing. It is of type `float`.

p   the ray parameter. It is of type `const float*`.

g   gradient of slowness (`const float*`).

sp   distance traveled in the medium (cell) times the velocity of the medium (equivalent to the optical path length in optics). It is of type `float*`.

jp   the direction of the ray. It is of type `int*`.

13.2.5 sf_cell11_update1

Performs the first step of the first order non-symplectic method for ray tracing.

Call

```
tt = sf_cell11_update1 (dim, s, v, p, g);
```

Definition

```
float sf_cell11_update1 (int dim, float s, float v, float *p, const float *g)
/*< nonsymplectic first-order: step 1 >*/
{
   ...
}
```

Input parameters

dim   dimension (`int`).

s σ (`float`).

v   slowness. It is of type `float`.

`p`	direction. It is of type `float*`.
`g`	slowness gradient. It is of type `const float*`.

Output

`0.5*v*v*s*(1. + s*pg)`	travel time. It is of type `float`.

13.2.6 sf_cell11_update2

Performs the second step of the first order non-symplectic method for ray tracing.

Call

```
tt = sf_cell11_update2 (dim, s, v, p, g);
```

Definition

```
float sf_cell11_update2 (int dim, float s, float v, float *p, const float *g)
/*< nonsymplectic first-order: step 2 >*/
{
   ...
}
```

Input parameters

`dim`	dimension (`int`).
`s`	σ (`float`).
`v`	slowness. It is of type `float`.
`p`	direction. It is of type `float*`.
`g`	slowness gradient. It is of type `const float*`.

Output

`0.5*v*v*s*(1. - s*pg)`	travel time. It is of type `float`.

13.2.7 sf_cell_intersect

Intersects a parabolic ray with the cell boundary.

Call

```
sf_cell_intersect (a, x, dy, p, sx, jx);
```

Definition

```
void sf_cell_intersect (float a, float x, float dy, float p,
                        float *sx, int *jx)
/*< intersecting a parabolic ray with cell boundaries >*/
{
   ...
}
```

Input parameters

`a` gradient of slowness (`float`).

`x` non-integer part of the position in the grid relative to grid origin. It is of type `float`.

`dy` depth or lateral sampling divided by slowness. It is of type `float`.

`p` the ray parameter. It is of type `float`.

`sx` distance traveled in the medium (cell) times the velocity of the medium (equivalent to the optical path length in optics). It is of type `float*`.

`jx` the direction of the ray. It is of type `int*`.

13.2.8 sf_cell_snap

Terminates the ray at the nearest boundary.

Definition

```
 b = sf_cell_snap (z, iz, eps);
```

Definition

```
bool sf_cell_snap (float *z, int *iz, float eps)
/*< round to the nearest boundary >*/
{
   ...
}
```

Input parameters

`z`	position (`float*`).
`iz`	sampling (`int*`).
`eps`	tolerance. It is of type `float`.

Output

`true/false` whether the ray is terminated or not. It is of type `bool`.

13.2.9 sf_cell_update1

Performs the first step of the second order symplectic method for ray tracing.

Call

```
tt = sf_cell_update1 (dim, s, v, p, g);
```

Definition

```
float sf_cell_update1 (int dim, float s, float v, float *p, const float *g)
/*< symplectic second-order: step 1 >*/
{
   ...
}
```

Input parameters

`dim`	dimension (`int`).
`s`	σ (`float`).
`v`	slowness. It is of type `float`.
`p`	direction. It is of type `float*`.
`g`	slowness gradient. It is of type `const float*`.

Output

`0.5*v*v*s*(1.  + s*pg)`	travel time. It is of type `float`.

13.2.10 sf_cell_update2

Performs the second step of the second order symplectic method for ray tracing.

Call

```
tt = sf_cell_update2 (dim, s, v, p, g);
```

Definition

```
float sf_cell_update2 (int dim        /* number of dimensions */,
                       float s        /* sigma */,
                       float v        /* slowness */,
                       float *p       /* in - ?, out - direction */,
                       const float *g /* slowness gradient */)
/*< symplectic second-order: step 2 >*/
{
   ...
}
```

Input parameters

`dim`	dimension (`int`).
`s`	σ (`float`).
`v`	slowness. It is of type `float`.
`p`	direction. It is of type `float*`.
`g`	slowness gradient. It is of type `const float*`.

Output

0.5*v*v*s*(1. - s*pg)	travel time. It is of type `float`.

13.2.11 sf_cell_p2a

Converts the ray parameter to an angle.

Call

```
a = sf_cell_p2a (p);
```

Definition

```
float sf_cell_p2a (float* p)
/*< convert ray parameter to angle >*/
{
   ...
}
```

Input parameters

`p`	the ray parameter (`float*`).

Output

`a`	angle of the ray. It is of type `float*`.

Chapter 14

General tools

14.1 First derivative FIR filter (deriv.c)

14.1.1 sf_deriv_init

Initializes the derivative calculation of the input trace, that is, it sets the required parameters and allocates the required space.

Call

```
sf_deriv_init (nt1, n1, c1);
```

Definition

```
void sf_deriv_init(int nt1  /* transform length */,
                   int n1   /* trace length */,
                   float c1 /* filter parameter */)
{
   ...
}
```

Input parameters

`nt1` length of the transform (derivative) (`int`).

`n1` length of the trace (`int`).

`c1` filter parameter (`float`).

14.1.2 sf_deriv_free

Frees the temporary space allocated for the derivative operator.

Call

```
sf_deriv_free ();
```

Definition

```
void sf_deriv_free(void)
{
   ...
}
```

14.1.3 sf_deriv

Calculates the derivative of the input trace (`trace`) and outputs it to `trace2`.

Definition

```
sf_deriv (trace, trace2);
```

Definition

```
void sf_deriv (const float* trace, float* trace2)
/*< derivative operator >*/
{
   ...
}
```

Input parameters

`trace` input trace whose derivative is required (`float*`).

`trace2` location where the derivative is to be stored (`float*`).

14.2 Computing quantiles by Hoare's algorithm (quantile.c)

14.2.1 sf_quantile

Returns the quantile - which is specified in the input - for the input array.

Call

```
k = sf_quantile(q, n, a);
```

Definition

```
float sf_quantile(int q    /* quantile */,
                  int n    /* array length */,
                  float* a /* array [n] */)
/*< find quantile (caution: a is changed) >*/
{
   ...
}
```

Input parameters

`q` the required quantile (`int`).

`n` length of the input array (`int`).

`a` the input array for which the quantile is required (`float*`).

Output

`*k` the quantile. It is of type `float`.

14.3 Pseudo-random numbers: uniform and normally distributed (randn.c)

14.3.1 sf_randn1

Generates a normally distributed random number using the Box-Muller method.

Call

```
vset = sf_randn_one_bm ();
```

Definition

```
float sf_randn_one_bm (void)
/*< return a random number (normally distributed, Box-Muller method) >*/
{
   ...
}
```

Output

`vset` the random number. It is of type `float`.

14.3.2 sf_randn

Fills an array with normally distributed random numbers.

Call

```
sf_randn (nr, r);
```

Definition

```
void sf_randn (int nr, float *r /* [nr] */)
/*< fill an array with normally distributed numbers >*/
{
   ...
}
```

Input parameters

`nr` size of the array where the random numbers are to be stored (`int`).

`r` the array where the random numbers are to be stored (`float*`).

14.3.3 sf_random

Fills an array with uniformly distributed random numbers.

Call

```
sf_random (nr, r)
```

Definition

```
void sf_random (int nr, float *r /* [nr] */)
/*< fill an array with uniformly distributed numbers >*/
{
   ...
}
```

Input parameters

`nr` size of the array where the random numbers are to be stored (`int`).

`nr` the array where the random numbers are to be stored (`float*`).

14.4 Evaluating mathematical expressions (math1.c)

14.4.1 myabs

Returns a complex number with zero imaginary value and the real non-zero real part is the absolute value of the input complex number.

Call

```
c = sf_complex myabs(c);
```

Definition

```
static sf_complex myabs(sf_complex c)
{
   ...
}
```

Input parameters

`c` a complex number (`sf_complex`).

Output

`c` a complex number with real part = absolute value of the input complex number and imaginary part = zero. It is of type `static sf_complex`.

14.4.2 myconj

Returns the complex conjugate of the input complex number.

Call

```
c = myconj(sf_complex c);
```

Definition

```
static sf_complex myconj(sf_complex c)
{
   ...
}
```

Input parameters

`c` a complex number (`sf_complex`).

Output

`c` complex conjugate of the input complex number.

14.4.3 myarg

Returns the argument of the input complex number.

Call

```
c = myarg(c);
```

Definition

```
static sf_complex myarg(sf_complex c)
{
   ...
}
```

Input parameters

`c` a complex number (`sf_complex`).

Output

`c` argument of the input complex number.

14.4.4 sf_math_evaluate

Applies a mathematical function to the input stack. For example it could evaluate the exponents of the samples in the stack.

Call

```
sf_math_evaluate(len, nbuf, fbuf, fst);
```

Definition

```
void sf_math_evaluate (int     len  /* stack length */,
                       int     nbuf /* buffer length */,
                       float** fbuf /* number buffers */,
                       float** fst  /* stack */)
/*< Evaluate a mathematical expression from stack (float numbers) >*/
{
   ...
}
```

Input parameters

`len`	length of the stack (`int`).
`nbuf`	length of the buffer (`int`).
`fbuf`	buffers for floating point numbers (`float**`).
`fst`	the stack (`float**`).

14.4.5 sf_complex_math_evaluate

Applies a mathematical function to the input stack. For example it could evaluate the exponents of the samples in the stack, It works like `sf_math_evaluate` but does it for complex numbers.

Call

```
sf_complex_math_evaluate(len, nbuf, cbuf, cst);
```

Definition

```
void sf_complex_math_evaluate (int          len  /* stack length */,
                               int          nbuf /* buffer length */,
                               sf_complex** cbuf /* number buffers */,
                               sf_complex** cst  /* stack */)
/*< Evaluate a mathematical expression from stack (complex numbers) >*/
{
   ...
}
```

Call

```
sf_complex_math_evaluate(len, nbuf, cbuf, cst)
```

Definition

```
void sf_complex_math_evaluate (int          len  /* stack length */,
                               int          nbuf /* buffer length */,
                               sf_complex** cbuf /* number buffers */,
                               sf_complex** cst  /* stack */)
/*< Evaluate a mathematical expression from stack (complex numbers) >*/
{
   ...
}
```

Input parameters

`len`	length of the stack (`int`).
`nbuf`	length of the buffer (`int`).
`fbuf`	buffers for floating point numbers (`sf_complex**`).
`fst`	the stack (`sf_complex**`).

14.4.6 sf_math_parse

Parses the mathematical expression and returns the stack length.

call

```
len = sf_math_parse(output, out, datatype);
```

Definition

```
size_t sf_math_parse (char*       output /* expression */,
                      sf_file     out    /* parameter file */,
                      sf_datatype datatype)
/*< Parse a mathematical expression, returns stack length >*/
{
   ...
}
```

Input parameters

`output`	the expression which is to be parsed (`char`).
`out`	parameter file (`sf_file`).
`datatype`	file datatype (`sf_datatype`).

Output

`len`	length of the stack. It is of type `size_t`.

14.4.7 sf_math_parse

Checks for any syntax errors.

Call

```
check();
```

Definition

```
static void check (void)
{
   ...
}
```

Chapter 15

Geometry

15.1 Construction of points (point.c)

15.1.1 printpt2d

Prints the value and location of a 2D point (position vector).

Call

```
printpt2d(pt2d P);
```

Definition

```
void printpt2d(pt2d P)
/*< print point2d info  >*/
{
   ...
}
```

Input parameters

`P`	a point (position vector) (`pt3d`).

15.1.2 printpt3d

Prints the value and location of a 3D point (position vector).

Call

```
printpt3d(P);
```

Definition

```
void printpt3d(pt3d P)
/*< print point3d info  >*/
{
   ...
}
```

Input parameters

`P`	a point (position vector) (`pt3d`).

15.1.3 pt2dwrite1

Outputs a 1D array of 2D points to a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt2dwrite1(F, v, n1, k);
```

Definition

```
void pt2dwrite1(sf_file F, pt2d *v, size_t n1, int k)
/*< output point2d 1-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file to which the 1D array of 2D points is to be output (`sf_file`).
`v`	an array of 2D points which is to be output (`pt2d`).
`n1`	size of the array (`size_t`).
`k`	a number, which if equal to 3, indicates that the value of the 2D points must also be included (`int`).

15.1.4 pt2dwrite2

Outputs a 2D array of 2D points to a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt2dwrite2(F, v, n1, n2, k);
```

Definition

```
void pt2dwrite2(sf_file F, pt2d **v, size_t n1, size_t n2, int k)
/*< output point2d 2-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file to which the 2D array of 2D points is to be output (`sf_file`).
`v`	an array of 2D points which is to be output (`pt2d`).
`n1`	size of one axis of the 2D array (`size_t`).
`n2`	size of the other axis of the 2D array (`size_t`).
`k`	a number, which if equal to 3, indicates that the value of the 2D points must also be included (`int`).

15.1.5 pt3dwrite1

Outputs a 1D array of 3D points to a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt3dwrite1(F, v, n1, k);
```

Definition

```
void pt3dwrite1(sf_file F, pt3d *v, size_t n1, int k)
/*< output point3d 1-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file to which the 1D array of 3D points is to be output (`sf_file`).
`v`	an array of 1D points which is to be output (`pt3d`).
`n1`	size of one axis of the 3D array (`size_t`).
`k`	a number, which if equal to 4, indicates that the value of the 3D points must also be included (`int`).

15.1.6 pt3dwrite2

Outputs a 2D array of 3D points to a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt3dwrite2(F, v, n1, n2, k);
```

Definition

```
void pt3dwrite2(sf_file F, pt3d *v, size_t n1, size_t n2, int k)
/*< output point3d 2-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file to which the 2D array of 3D points is to be output (`sf_file`).
`v`	an array of 2D points which is to be output (`pt3d`).
`n1`	size of one axis of the 2D array (`size_t`).
`n2`	size of the other axis of the 2D array (`size_t`).
`k`	a number, which if equal to 4, indicates that the value of the 3D points must also be included (`int`).

15.1.7 pt2dread1

Reads a 1D array of 2D points from a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt2dread1(F, v, n1, k);
```

Definition

```
void pt2dread1(sf_file F, pt2d *v, size_t n1, int k)
/*< input point2d 1-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file from which the 1D array of 2D points is to be read (`sf_file`).
`v`	an array of 2D points which is to be read (`pt2d`).
`n1`	size of the array (`size_t`).
`k`	a number, which if equal to 3, indicates that the value of the 2D points must also be included (`int`).

15.1.8 pt2dread2

Reads a 2D array of 2D points from a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt2dread2(F, v, n1, n2, k);
```

Definition

```
void pt2dread1(sf_file F, pt2d *v, size_t n1, int k)
/*< input point2d 1-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file from which the 2D array of 2D points is to be read (`sf_file`).
`v`	an array of 2D points which is to be read (`pt2d`).
`n1`	size of one axis of the 2D array (`size_t`).
`n2`	size of the other axis of the 2D array (`size_t`).
`k`	a number, which if equal to 3, indicates that the value of the 2D points must also be included (`int`).

15.1.9 pt3dread1

Reads a 1D array of 3D points from a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt3dread1(F, v, n1, k);
```

Definition

```
void pt3dread1(sf_file F, pt3d *v, size_t n1, int k)
/*< input point3d 1-D vector >*/
{
   ...
}
```

Input parameters

`File`	a file from which the 1D array of 3D points is to be read (`sf_file`).
`v`	an array of 1D points which is to be read (`pt3d`).
`n1`	size of one axis of the 3D array (`size_t`).
`k`	a number, which if equal to 4, indicates that the value of the 3D points must also be included (`int`).

15.1.10 pt3dread2

Reads a 2D array of 3D points from a file. It can be used to define the source or receiver arrays, for example.

Call

```
pt3dread2(F, v, n1, n2, k);
```

Definition

```
void pt3dread2(sf_file F, pt3d **v, size_t n1, size_t n2, int k)
/*< input point3d 2-D vector >*/
{
    ...
}
```

Input parameters

`File`	a file from which the 2D array of 3D points is to be read (`sf_file`).
`v`	an array of 2D points which is to be read (`pt3d`).
`n1`	size of one axis of the 2D array (`size_t`).
`n2`	size of the other axis of the 2D array (`size_t`).
`k`	a number, which if equal to 4, indicates that the value of the 3D points must also be included (`int`).

15.1.11 pt2dalloc1

Allocates memory for 1D array of 2D points.

Call

```
ptr = pt2dalloc1(n1);
```

Definition

```
pt2d* pt2dalloc1( size_t n1)
/*< alloc point2d 1-D vector >*/
{
   ...
}
```

Input parameters

`n1`	size of the array (`size_t`).

Output

`ptr`	pointer to memory (`pt2d*`).

15.1.12 pt2dalloc2

Allocates memory for 2D array of 2D points.

Call

```
ptr = pt2dalloc2(n1,n2);
```

Definition

```
pt2d** pt2dalloc2( size_t n1, size_t n2)
/*< alloc point2d 2-D vector >*/
{
   ...
}
```

Input parameters

`n1`	size one axis of the 2D array (`size_t`).
`n2`	size of the other axis of the 2D array (`size_t`).

Output

`ptr`	pointer to memory (`pt2d**`).

15.1.13 pt2dalloc3

Allocates memory for 3D array of 2D points.

Call

```
pt2d*** pt2dalloc3(n1, n2, n3);
```

Definition

```
pt2d*** pt2dalloc3(size_t n1, size_t n2, size_t n3)
/*< alloc point2d 3-D vector >*/
{
    ...
}
```

Input parameters

`n1`	size of first axis of the 3D array (`size_t`).
`n2`	size of the second axis of the 3D array (`size_t`).
`n3`	size of the third axis of the 3D array (`size_t`).

Output

`ptr`	pointer to memory (`pt2d***`).

15.1.14 pt3dalloc1

Allocates memory for 1D array of 3D points.

Call

```
ptr = pt3dalloc1(n1);
```

Definition

```
pt3d* pt3dalloc1( size_t n1)
/*< alloc point3d 1-D vector >*/
{
   ...
}
```

Input parameters

`n1` size of the array (`size_t`).

Output

`ptr` pointer to memory (`pt3d*`).

15.1.15 pt3dalloc2

Allocates memory for 2D array of 3D points.

Call

```
ptr = pt3dalloc2(n1,n2);
```

Definition

```
pt3d** pt3dalloc2( size_t n1, size_t n2)
/*< alloc point3d 2-D vector >*/
{
   ...
}
```

Input parameters

`n1` size of one axis of the 2D array (`size_t`).

`n2` size of the other axis of the 2D array (`size_t`).

Output

`ptr` pointer to memory (`pt3d**`).

15.1.16 pt3dalloc3

Allocates memory for 3D array of 3D points.

Call

```
ptr =  pt3dalloc3(n1, n2, n3);
```

Definition

```
pt3d*** pt3dalloc3(size_t n1, size_t n2, size_t n3)
/*< alloc point3d 3-D vector >*/
{
   ...
}
```

Input parameters

`n1`	size of first axis of the 3D array (`size_t`).
`n2`	size of the second axis of the 3D array (`size_t`).
`n3`	size of the third axis of the 3D array (`size_t`).

Output

`ptr`	pointer to memory (`pt3d***`).

15.2 Construction of vectors (vector.c)

15.2.1 det3

The determinant of a 3×3 matrix.

Call

```
d = det3(m);
```

Definition

```
double det3(double *m)
{
   ...
}
```

Input parameters

`m`	a 3×3 matrix (`double`).

Output

`d`	the determinant (`double`).

15.2.2 det2

The determinant of a 2×2 matrix.

Call

```
d = det2(m);
```

Definition

```
double det2(double *m)
{
   ...
}
```

Input parameters

`m` a 2×2 matrix (`double`).

Output

`d` the determinant (`double`).

15.2.3 jac3d

Returns a 3D jacobian.

Call

```
r = jac3d(C, T, P, Q);
```

Definition

```
double jac3d(pt3d *C, pt3d *T, pt3d *P, pt3d *Q)
/*< 3D jacobian >*/
{
   ...
}
```

Input parameters

`c` a complex number. Must be of type `sf_double_complex`.

Output

`c.r` real part of the complex number. It is of type `double`.

15.2.4 vec3d

Builds a 3D vector. The components of the vector returned are the difference of the respective components of the two input points (position vectors). The first input vector is the origin.

Call

```
V = vec3d(O, A);
```

Definition

```
vc3d vec3d(pt3d* O, pt3d* A)
/*< build 3D vector >*/
{
   ...
}
```

Input parameters

`O` a 3D point, this serves as the origin (`pt3d`).

`A` a 3D point (`pt3d`).

Output

`V` the 3D vector. It is of type `vc3d`.

15.2.5 axa3d

Builds a 3D unit vector. The components of the vector returned are zero except for the one indicated in the input argument `n`, which is equal to one. If `n`=1 the z axis is 1, if `n`=2 the x axis is 1 and if `n`=3 the y axis is 1.

Call

```
V = axa3d (n);
```

Definition

```
vc3d axa3d( int n)
/*< build 3D unit vector >*/
{
   ...
}
```

Input parameters

`n` a number which indicates which axis is to be set equal to 1 (`int`).

Output

`V` the 3D unit vector. It is of type `vc3d`.

15.2.6 scp3d

Returns the scalar product of the two 3D vectors.

Call

```
p = scp3d(U, V);
```

Definition

```
double scp3d(vc3d* U, vc3d* V)
/*< scalar product of 3D vectors >*/
{
   ...
}
```

Input parameters

`U`	a 3D vector (`vc3d`).
`V`	a 3D vector (`vc3d`).

Output

`V->dx*V->dx + V->dy*V->dy + V->dz*V->dz`	the 3D unit vector.

15.2.7 vcp3d

Returns the vector product of the two input vectors.

Call

```
W = vcp3d(U, V);
```

Definition

```
vc3d vcp3d(vc3d* U, vc3d* V)
/*< vector product of 3D vectors >*/
{
   ...
}
```

Input parameters

`U`	a 3D vector (`vc3d`).
`V`	a 3D vector (`vc3d`).

Output

`W`	the 3D unit vector. It is of type `vc3d`.

15.2.8 len3d

Returns the length of a 3D vector.

Call

```
l = len3d(V);
```

Definition

```
double len3d(vc3d* V)
/*< 3D vector length >*/
{
   ...
}
```

Input parameters

`V` a 3D vector (`vc3d`).

Output

`l` the length of the 3D vector. It is of type `double`.

15.2.9 nor3d

Normalizes a 3D vector. The components of the 3D input vector are divided by its length.

Call

```
W = nor3d(V);
```

Definition

```
vc3d nor3d(vc3d* V)
/*< normalize 3D vector >*/
{
   ...
}
```

Input parameters

`V` the input 3D vector. It is of type `vc3d`.

Output

`W` the normalized 3D vector. It is of type `vc3d`.

15.2.10 ang3d

Returns the angle between the input 3D vectors.

Call

```
a = ang3d(U, V);
```

Definition

```
double ang3d(vc3d* U, vc3d* V)
/*< angle between 3D vectors >*/
{
   ...
}
```

Input parameters

`U` the input 3D vector. It is of type `vc3d`.

`V` the input 3D vector. It is of type `vc3d`.

Output

`a` the angle between the two input 3D vectors. It is of type `double`.

15.2.11 tip3d

Returns the tip of a 3D vector. The components of the vector returned are the sum of the respective components of the two input points (position vectors). Unlike the `sf_vc3d` where the first input vector was the origin, in `sf_tip3d` the origin is zero. This means that the vector returned is a position vector or simply a point, which is of type `pt3d`.

Call

```
A = tip3d(O, V);
```

Definition

```
pt3d tip3d(pt3d* O, vc3d* V)
/*< tip of a 3D vector >*/
{
   ...
}
```

Input parameters

`O` a 3D point (`pt3d`).

`V` a 3D point (`pt3d`).

Output

`V` the 3D vector. It is of type `vc3d`.

15.2.12 scl3d

Scales a 3D vector, that is, it multiplies it by a scalar. The components of the vector returned are the product of the components of the input vector and the input scalar.

Call

```
W = scl3d(V, s);
```

Definition

```
vc3d scl3d(vc3d* V, float s)
/*< scale a 3D vector >*/
{
   ...
}
```

Input parameters

`V` a 3D point (`pt3d`).

`s` a scalar which is to be multiplied by every component of the input vector (`float`).

Output

`W` the scaled 3D vector. It is of type `vc3d`.

15.3 Conversion between line and Cartesian coordinates of a vector (decart.c)

15.3.1 sf_line2cart

Converts the line coordinates to Cartesian coordinates.

Call

```
sf_line2cart(dim, nn, i, ii);
```

Definition

```
void sf_line2cart(int dim       /* number of dimensions */,
                  const int* nn /* box size [dim] */,
                  int i         /* line coordinate */,
                  int* ii       /* cartesian coordinates [dim] */)
/*< Convert line to Cartesian >*/
{
   ...
}
```

Input parameters

`dim` number of dimensions (`int`).

`nn` box size (size of the data file) (`const int*`).

`i` the line coordinate (`int`).

`ii` the Cartesian coordinates (`int*`).

15.3.2 sf_cart2line

Converts the Cartesian coordinates to line coordinate.

Call

```
int sf_cart2line(dim, nn, ii);
```

Definition

```
int sf_cart2line(int dim       /* number of dimensions */,
                 const int* nn /* box size [dim] */,
                 const int* ii /* cartesian coordinates [dim] */)
/*< Convert Cartesian to line >*/
{
   ...
}
```

Input parameters

`dim`	number of dimensions (`int`).
`nn`	box size (size of the data file) (`const int*`).
`ii`	the Cartesian coordinates (`int*`).

Output

`i`	line coordinate. It is of type `int`.

15.3.3 sf_first_index

Returns the first index for a particular dimension.

Call

```
sf_first_index (i, j, dim, n, s);
```

Definition

```
int sf_first_index (int i        /* dimension [0...dim-1] */,
                    int j        /* line coordinate */,
                    int dim      /* number of dimensions */,
                    const int *n /* box size [dim] */,
                    const int *s /* step [dim] */)
/*< Find first index for multidimensional transforms >*/
{
   ...
}
```

Input parameters

`i`	the dimension (`int`).
`j`	the line coordinate (`int`).
`dim`	number of dimensions (`int`).
`n`	box size (size of the data file) (`const int*`).
`s`	the step size (`const int*`).

Output

`i0` first index for the given dimension. It is of type `int`.

15.3.4 sf_large_line2cart

Converts the line coordinate to Cartesian coordinates. It works exactly like `sf_line2cart` but in this one the line and Cartesian coordinates are of type `off_t`, which means that they are given in terms of the offset in bytes in the data file.

Call

```
sf_large_line2cart(dim, nn, i, ii);
```

Definition

```
void sf_large_line2cart(int dim         /* number of dimensions */,
                        const off_t* nn /* box size [dim] */,
                        off_t i         /* line coordinate */,
                        off_t* ii       /* cartesian coordinates [dim] */)
/*< Convert line to Cartesian >*/
{
   ...
}
```

Input parameters

`dim`	number of dimensions (`int`).
`nn`	box size (size of the data file) (`const off_t*`).
`i`	the line coordinate (`off_t`).
`ii`	the Cartesian coordinates (`off_t*`).

15.3.5 sf_large_cart2line

Converts the Cartesian coordinates to line coordinate. It works exactly like `sf_line2cart` but in this one the line and Cartesian coordinates are of type `off_t`, which means that they are given in terms of the offset in bytes in the data file.

Call

```
sf_large_cart2line(int, nn, ii);
```

Definition

```
off_t sf_large_cart2line(int dim         /* number of dimensions */,
                         const off_t* nn /* box size [dim] */,
                         const off_t* ii /* cartesian coordinates [dim] */)
/*< Convert Cartesian to line >*/
{
   ...
}
```

Input parameters

`dim`	number of dimensions (`int`).
`nn`	box size (size of the data file) (`const off_t*`).
`ii`	the Cartesian coordinates (`const off_t*`).

Output

`i` line coordinate. It is of type `off_t`.

15.3.6 sf_large_first_index

Returns the first index for a particular dimension. It works exactly like `sf_first_index` but in this one the line coordinate, box size, step size and the first index are of type `off_t`, which means that they are given in terms of the offset in bytes in the data file.

Call

```
sf_large_first_index (i, j, dim, n, s);
```

Definition

```
off_t sf_large_first_index (int i          /* dimension [0...dim-1] */,
                            off_t j        /* line coordinate */,
                            int dim        /* number of dimensions */,
                            const off_t *n /* box size [dim] */,
                            const off_t *s /* step [dim] */)
/*< Find first index for multidimensional transforms >*/
{
   ...
}
```

Input parameters

`i`	the dimension (`int`).
`j`	the line coordinate (`off_t`).
`dim`	number of dimensions (`int`).
`n`	box size (size of the data file) (`const off_t*`).
`s`	the step size (`const off_t*`).

Output

`i0` first index for the given dimension. It is of type `int`.

15.4 Axes (axa.c)

15.4.1 sf_maxa

Creates a simple axis.

Call

```
AA = sf_maxa(n, o, d);
```

Definition

```
sf_axis sf_maxa(int n   /* length */,
                float o /* origin */,
                float d /* sampling */)
/*< make a simple axis >*/
{
   ...
}
```

Input parameters

`n` length of the axis (`int`).

`o` origin of the axis (`float`).

`d` sampling of the axis (`float`).

Output

`AA` the axis. It is of type `sf_axis`.

15.4.2 sf_iaxa

Reads an axis from the file which is given in the input.

Call

```
AA = sf_iaxa(FF, i);
```

Definition

```
sf_axis sf_iaxa(sf_file FF, int i)
/*< read axis i >*/
{
   ...
}
```

Input parameters

`FF` the file from which the axis is to be read (`sf_file`).

`i` a number which specified which axis is to be read, for example `n1`, `n2`, `n3` etc (`int`).

Output

`AA` location where the axis is stored. It is of type `sf_axis`.

15.4.3 sf_oaxa

Writes an axis, from the input location, to the file, which is also given in the input.

Call

```
sf_oaxa(FF, AA, i);
```

Definition

```
void sf_oaxa(sf_file FF, const sf_axis AA, int i)
/*< write axis i >*/
{
   ...
}
```

Input parameters

`FF`	the file in which the axis is to be written (`sf_file`).
`AA`	the location from where the axis is to be read. Must be of type `const sf_axis`.
`i`	a number which specified which axis is to be read, for example `n1`, `n2`, `n3` etc (`int`).

15.4.4 sf_raxa

Prints the information about the axis on the screen.

Call

```
sf_raxa(AA);
```

Definition

```
void sf_raxa(const sf_axis AA)
/*< report information on axis AA >*/
{
   ...
}
```

Input parameters

`AA`	the axis about which the information is required. Must be of type `const sf_axis`.

15.4.5 sf_n

Provides access to the length of the axis.

Call

```
AA->n = sf_n(AA);
```

Definition

```
int sf_n(const sf_axis AA)
/*< access axis length >*/
{
   ...
}
```

Input parameters

`AA` the axis whose length is to be accessed. Must be of type `const sf_axis`.

Output

`AA->n` length of the axis. It is of type `int`.

15.4.6 sf_o

Provides access to the origin of the axis.

Call

```
AA->o = sf_o(AA);
```

Definition

```
float sf_o(const sf_axis AA)
/*< access axis origin >*/
{
   ...
}
```

Input parameters

`AA` the axis whose length is to be accessed. Must be of type `const sf_axis`.

Output

`AA->o` length of the axis. It is of type `float`.

15.4.7 sf_d

Provides access to the sampling of the axis.

Call

```
AA->d = sf_d(AA);
```

Definition

```
float sf_d(const sf_axis AA)
/*< access axis sampling >*/
{
   ...
}
```

Input parameters

`AA` the axis whose length is to be accessed. Must be of type `const sf_axis`.

Output

`AA->d` length of the axis. It is of type `float`.

15.4.8 sf_nod

Copies the length, origin and sampling of the input axis to another place which is also an object of type `sf_axis`.

Call

```
BB = sf_nod(AA);
```

Definition

```
sf_axa sf_nod(const sf_axis AA)
/*< access length, origin, and sampling >*/
{
   ...
}
```

Input parameters

`AA` the axis whose length, origin and sampling is to be accessed. Must be of type `const sf_axis`.

Output

`BB` the location where the length, origin and sampling are copied. It is of type `sf_axis`.

15.4.9 sf_setn

Changes the length of the axis.

Call

```
AA->n = sf_setn(AA, n);
```

Definition

```
void sf_setn(sf_axis AA, int n)
/*< change axis length >*/
{ AA->n=n; }
```

Input parameters

`AA`	the axis whose length is to be changed (`sf_axis`).
`n`	the new length which is to be set (`int`).

15.4.10 sf_seto

Changes the origin of the axis.

Call

```
AA->o = sf_seto(AA, o);
```

Definition

```
void sf_seto(sf_axis AA, float o)
/*< change axis origin >*/
{
   ...
}
```

Input parameters

`AA`	the axis whose origin is to be changed (`sf_axis`).
`o`	the new origin which is to be set (`float`).

15.4.11 sf_setd

Changes the sampling of the axis.

Call

```
AA->d = sf_setd(AA, d);
```

Definition

```
void sf_setd(sf_axis AA, float d)
/*< change axis sampling >*/
{
   ...
}
```

Input parameters

`AA`	the axis whose sampling is to be changed (`sf_axis`).
`o`	the new sampling which is to be set (`float`).

15.4.12 sf_setlabel

Changes the label of the axis.

Call

```
sf_setlabel(AA, label);
```

Definition

```
void sf_setlabel(sf_axis AA, const char* label)
/*< change axis label >*/
{
   ...
}
```

Input parameters

`AA`	the axis whose label is to be changed (`sf_axis`).
`label`	the new label which is to be set (`const char*`).

15.4.13 sf_setunit

Changes the unit of the axis.

Call

```
sf_setunit(AA, unit);
```

Definition

```
void sf_setunit(sf_axis AA, const char* unit)
/*< change axis unit >*/
{
   ...
}
```

Input parameters

`AA`	the axis whose unit is to be changed (`sf_axis`).
`unit`	the new unit which is to be set (`const char*`).

Chapter 16

Miscellaneous

16.1 sharpening (sharpen.c)

16.1.1 sf_sharpen_init

Initializes the sharpening operator by allocating the required and initializing the required operators.

Call

```
sf_sharpen_init(n1, perc);
```

Definition

```
void sf_sharpen_init(int n1     /* data size */,
                     float perc /* quantile percentage */)
/*< initialize >*/
{
   ...
}
```

Input parameters

`n1`	size of the data (`int`).
`perc`	the quantile percentage (`float`).

16.1.2 sf_sharpen_close

Frees the allocated memory for the sharpening calculation.

Call

```
sf_sharpen_close();
```

Definition

```
void sf_sharpen_close(void)
/*< free allocated storage >*/
{
   ...
}
```

16.1.3 sf_sharpen

Computes the weights for the sharpening regularization.

Call

```
wp = sf_sharpen(pp);
```

Definition

```
float sf_sharpen(const float *pp)
/*< compute weight for sharpening regularization >*/
{
   ...
}
```

Input parameters

`pp`	an array for which the weights are to be calculated (`const float*`).

Output

`wp`	weights for sharpening regularization. It is of type `float`.

16.1.4 sf_csharpen

Computes the weights for the sharpening regularization for complex numbers.

Call

```
sf_csharpen(pp);
```

Definition

```
void sf_csharpen(const sf_complex *pp)
/*< compute weight for sharpening regularization >*/
{
   ...
}
```

Input parameters

`pp`	an array for which the weights are to be calculated (`sf_complex`).

16.2 Sharpening inversion added Bregman iteration (sharpinv.c)

16.2.1 sf_csharpinv

Performs the sharp inversion to estimate the model from the data, for complex numbers.

Call

```
sf_csharpinv(oper, scale, niter, ncycle, perc, verb, nq, np, qq, pp);
```

Definition

```
void sf_csharpinv(sf_coperator oper /* inverted operator */,
                  float scale       /* extra operator scaling */,
                  int niter         /* number of outer iterations */,
                  int ncycle        /* number of iterations */,
                  float perc        /* sharpening percentage */,
                  bool verb         /* verbosity flag */,
                  int nq, int np    /* model and data size */,
                  sf_complex *qq    /* model */,
                  sf_complex *pp    /* data */)
/*< sharp inversion for complex-valued operators >*/
{
   ...
}
```

Input parameters

`oper`	the inverted operator (`sf_operator`).
`scale`	extra operator scaling (`float`).
`niter`	number of outer iterations (`int`).
`ncycle`	number of iterations (`int`).
`perc`	sharpening percentage (`float`).
`verb`	verbosity flag (`bool`).
`nq`	size of the model (`int`).
`np`	size of the data (`int`).
`qq`	the model. Must be of type textttsf_complex*.
`pp`	the data (`sf_complex*`).

16.2.2 sf_sharpinv

Performs the sharp inversion to estimate the model from the data.

Call

```
void sf_sharpinv(oper, scale, niter, ncycle, perc, verb, nq, np, qq, pp);
```

Definition

```
void sf_sharpinv(sf_operator oper  /* inverted operator */,
                 float scale       /* extra operator scaling */,
                 int niter         /* number of outer iterations */,
                 int ncycle        /* number of iterations */,
                 float perc        /* sharpening percentage */,
                 bool verb         /* verbosity flag */,
                 int nq, int np    /* model and data size */,
                 float *qq         /* model */,
                 float *pp         /* data */)
/*< sharp inversion for real-valued operators >*/
{
   ...
}
```

Input parameters

`oper`	the inverted operator (`sf_operator`).
`scale`	extra operator scaling (`float`).
`niter`	number of outer iterations (`int`).
`ncycle`	number of iterations (`int`).
`perc`	sharpening percentage (`float`).
`verb`	verbosity flag (`bool`).
`nq`	size of the model (`int`).
`np`	size of the data (`int`).
`qq`	the model (`float*`).
`pp`	the data (`float*`).

Chapter 17

System

17.1 Priority queue (heap sorting) (pqueue.c)

17.1.1 sf_pqueue_init

Initializes the heap with the maximum size given in the input.

Call

```
sf_pqueue_init (n);
```

Definition

```
void sf_pqueue_init (int n)
/*< Initialize heap with the maximum size >*/
{
   ...
}
```

Input parameters

`n`	maximum size of the heap (`int`).

17.1.2 sf_pqueue_start

Sets the starting values for the queue.

Call

```
sf_pqueue_start ();
```

Definition

```
void sf_pqueue_start (void)
/*< Set starting values >*/
{
   ...
}
```

17.1.3 sf_pqueue_close

Frees the space allocated by `sf_pqueue_init`.

Call

```
sf_pqueue_close();
```

Definition

```
void sf_pqueue_close (void)
/*< Free the allocated storage >*/
{
   ...
}
```

17.1.4 sf_pqueue_insert

Inserts an element in the queue. The smallest element goes first.

Call

```
sf_pqueue_insert (v);
```

Definition

```
void sf_pqueue_insert (float* v)
/*< Insert an element (smallest first) >*/
{
   ...
}
```

Input parameters

v	element to be inserted, smallest first (`float*`).

17.1.5 sf_pqueue_insert2

Inserts an element in the queue. The largest element goes first.

Call

```
sf_pqueue_insert2 (v);
```

Definition

```
void sf_pqueue_insert2 (float* v)
/*< Insert an element (largest first) >*/
{
   ...
}
```

Input parameters

v	element to be inserted, largest first (`float*`).

17.1.6 sf_pqueue_extract

Extracts the smallest element from the list.

Call

```
v = sf_pqueue_extract();
```

Definition

```
float* sf_pqueue_extract (void)
/*< Extract the smallest element >*/
{
    unsigned int c;
    int n;
   ...
    return v;
}
```

Output

v	the extracted smallest element (`float*`).

17.1.7 sf_pqueue_extract2

Extracts the largest element from the list.

Call

```
v = sf_pqueue_extract2();
```

Definition

```
float* sf_pqueue_extract2 (void)
/*< Extract the largest element >*/
{
   ...
}
```

Output

`v`	the extracted largest element (`float*`).

17.1.8 sf_pqueue_update

Updates the heap.

Call

```
sf_pqueue_update (v);
```

Definition

```
void sf_pqueue_update (float **v)
/*< restore the heap: the value has been altered >*/
{
   ...
}
```

Input parameters

`v`	elements to be inserted, largest first (`float**`).

17.2 Simplified system command (system.c)

17.2.1 sf_system

Runs a system command given to it as an input.

Call

```
sf_system(command);
```

Definition

```
void sf_system(const char *command)
/*< System command >*/
{
   ...
}
```

Input parameters

`command`	the command which is to be run on the system (`const char`).

17.3 Generic stack (FILO) structure operations (stack.c)

17.3.1 sf_stack_init

Initializes the object of type `sf_stack`, that is, it allocates the required memory for the data and also sets the size of the stack.

Call

```
s = sf_stack_init (size_t size);
```

Definition

```
sf_stack sf_stack_init (size_t size)
/*< create a stack >*/
{
   ...
}
```

Input parameters

`size`	size of the stack (`size_t`).

Output

`s`	a stack (an object of type `sf_stack`). It is of type `sf_stack`.

17.3.2 sf_stack_print

Prints the information about the stack on the screen. This may be used for debugging.

Call

```
sf_stack_print(s);
```

Definition

```
void sf_stack_print (sf_stack s)
/*< print out a stack (for debugging) >*/
{
   ...
}
```

Input parameters

`s`	a stack (an object of type `sf_stack`). It is of type `sf_stack`.

17.3.3 sf_stack_get

Extracts the length of the stack.

Call

```
l = sf_stack_get(s);
```

Definition

```
int sf_stack_get (sf_stack s)
/*< extract stack length >*/
{
   ...
}
```

Call

```
sf_stack_set(s, pos);
```

Definition

```
void sf_stack_set (sf_stack s, int pos)
/*< set stack position >*/
{
   ...
}
```

Input parameters

`s`	a stack (an object of type `sf_stack`). It is of type `sf_stack`.

Output

`s->top - s->entry`	length of the stack. It is of type `int`.

17.3.4 sf_stack_set

Sets the position of the pointer in the stack to the specified in the input `pos`.

```
sf_stack_set (s, pos);
```

```
void sf_stack_set (sf_stack s, int pos)
/*< set stack position >*/
{
   ...
}
```

Input parameters

`s`	a stack (an object of type `sf_stack`). It is of type `sf_stack`.
`pos`	desired position of the pointer in the stack. It is of type `int`.

17.3.5 sf_push

Inserts the data into the stack.

Call

```
sf_push(s, data, type);
```

Definition

```
void sf_push(sf_stack s, void *data, int type)
/*< push data into stack (requires unique data for each push) >*/
{
   ...
}
```

Input parameters

`s`	a stack (an object of type `sf_stack`). It is of type `sf_stack`.
`data`	data which is to be written into the stack. It is of type `void*`.
`type`	type of the data. It is of type `int`.

17.3.6 sf_pop

Extracts the data from the stack.

Call

```
dat = sf_pop(s);
```

Definition

```
void* sf_pop(sf_stack s)
/*< pop data from stack >*/
{
   ...
}
```

Input parameters

`s`	a stack (an object of type `sf_stack`). It is of type `sf_stack`.

Output

`old->data`	extracted data from the stack. It is of type `void*`.

17.3.7 sf_full

Tests whether the stack is full or not.

Call

```
 isfull = sf_full(s);
```

Definition

```
bool sf_full (sf_stack s)
/*< test if the stack is full >*/
{
   ...
}
```

Input parameters

s	a stack (an object of type `sf_stack`). It is of type `sf_stack`.

Output

`s->top >= s->entry`	true, if the stack is full, false otherwise. It is of type `bool`.

17.3.8 sf_top

Returns the data type of the top entry of the stack.

Call

```
typ = sf_top(s);
```

Definition

```
int sf_top(sf_stack s)
/*< return the top type >*/
{
   ...
}
```

Input parameters

s	a stack (an object of type `sf_stack`). It is of type `sf_stack`.

Output

`s->top->type`	type of the top entry. It is of type `int`.

17.3.9 sf_stack_close

Frees the space allocated for the stack.

Call

```
sf_stack_close(s);
```

Definition

```
void sf_stack_close(sf_stack s)
/*< free allocated memory >*/
{
   ...
}
```

Input parameters

`s` a stack (an object of type `sf_stack`). It is of type `sf_stack`.

Index