Chapter 6

Linear operators

6.1 Introduction

This section contains a bunch of programs that implement operators. Therefore a short introduction on operators is in order.

6.1.1 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 $\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]^T$ 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.

6.1.2 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 $L_1 L_2$ 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`.

6.1.3 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 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. $(L_1 L_2)^* = L_2^* L_1^*$. This is naturally extended to the product of any finite product of operators, i.e. $(L_1 L_2 \cdots L_n)^* = L_n^* L_{n-1}^* \cdots L_1^*$. The adjoint of the product is also implemented in `chain.c`.

[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.

6.1.4 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,

$$\langle a, Lb \rangle = \langle (L^* a)^*, b \rangle \tag{6.1}$$

where $\langle\, , \rangle$ denotes the dot product. Remember that the dot product of two functions $f, g \in \mathbb{L}_2$ is $\int f g^* \; dt$ while the dot product of two vectors $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{x}^H \mathbf{y}$. Notice that for vectors eq. (6.1) becomes $\mathbf{x}^H \mathbf{L} \mathbf{y} = (\mathbf{L}^H \mathbf{y})^H \mathbf{y}$ which is obviously true. The lhs of eq. (6.1) 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`.

6.1.5 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 6.1 summarizes the effect of the `adj` and `add` variables.

Table 6.1: 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	$x \leftarrow x + L^*y$

6.2 Adjoint zeroing (adjnull.c)

The null operator is defined by

$$y = 0x = 0, \qquad \text{with} \quad y_t \leftarrow 0.$$

Its adjoint is

$$x = 0^*y = 0, \qquad \text{with} \quad x_t \leftarrow 0.$$

6.2.1 sf_adjnull

Usage

```
sf_adjnull(adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + 0^*y$ is computed.

`nx` size of `x` (`int`).

`ny` size of `y` (`int`).

`x` input data or output (`float*`).

`y` output or input data (`float*`).

6.2.2 sf_cadjnull

The same as `sf_adjnull` but for complex data.

Usage

```
sf_cadjnull(adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + 0^* y$ is computed.

`nx` size of `x` (`int`).

`ny` size of `y` (`int`).

`x` input data or output (`sf_complex*`).

`y` output or input data (`sf_complex*`).

6.3 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

$$x = 1^* y = y, \qquad \text{with} \quad x_t \leftarrow y_t.$$

6.3.1 sf_copy_lop

Usage

```
sf_copy_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + 1^* y$ is computed.

`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*`).

6.4 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

$$x = 1^* y = y, \qquad \text{with} \quad x_t \leftarrow y_t.$$

6.4.1 sf_ccopy_lop

Usage

```
sf_ccopy_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + 1^* y$ is computed.

`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*`).

6.5 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

$$x = L_m^* y = my, \qquad \text{with} \quad x_t \leftarrow m_t y_t,$$

6.5.1 sf_mask_init

Initializes the static variable `m` with boolean values, to be used in the `sf_mask_lop` or `sf_cmask_lop`.

Usage

```
sf_mask_init (m)
```

Input parameters

`m` a pointer to boolean values (`const bool*`).

6.5.2 sf_mask_lop

Usage

```
sf_mask_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + L_m^* y$ is computed.

`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*`).

6.5.3 sf_cmask_lop

The same as `sf_mask_lop` but for complex data.

Usage

```
sf_cmask_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + L_m^* y$ is computed.

`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*`).

6.6 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.

$$x = L_w x = w^2 x, \qquad \text{with} \quad x_t \leftarrow w_t^2 x_t.$$

6.6.1 sf_weight_init

Initializes the weights to be applied as linear operator, by assigning value to a static parameter.

Usage

```
sf_weight_init(w)
```

Input parameters

`w` values of the weights (`float*`).

6.6.2 sf_weight_lop

Applies the linear operator with the weights initialized by `sf_weight_init`.

Usage

```
sf_weight_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + L_w^* y$ is computed.

`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*`).

6.6.3 sf_cweight_lop

The same as `sf_weight_lop` but for complex data.

Usage

```
sf_cweight_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + L_w^* y$ is computed.

`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*`).

6.6.4 sf_weight_apply

Creates a product of the weights squared and the input `x`.

Usage

```
sf_weight_apply (nx, x)
```

Input parameters

`nx` size of `x` (`int`).

`x` input data and output (`float*`).

6.6.5 sf_cweight_apply

The same as the `sf_weight_apply` but for the complex numbers.

Usage

```
sf_cweight_apply (nx, x)
```

Input parameters

`nx` size of `x` (`int`).

`x` input data and output (`sf_complex*`).

6.7 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

$$x = D^* y, \qquad \text{with} \quad x_t \leftarrow -(y_t - y_{t-1}),\ x_0 = -y_0.$$

6.7.1 sf_igrad1_lop

Usage

```
sf_igrad1_lop(adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + D^* y$ is computed.

`nx` size of `x` (`int`).

`ny` size of `y` (`int`).

`x` input data or output (`float*`).

`y` output or input data (`float*`).

6.8 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.

6.8.1 sf_causint_lop

Usage

```
sf_causint_lop (adj, add, nx, ny, x, y)
```

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 $x \leftarrow x + L^* y$ is computed.

`nx` size of `x` (`int`).

`ny` size of `y` (`int`).

`x` input data or output (`float*`).

`y` output or input data (`float*`).

6.9 Chaining linear operators (chain.c)

Calculates products of operators

6.9.1 sf_chain

Chains two operators L_1 and L_2:

$$d = (L_2 L_1) m.$$

Its adjoint is

$$m = (L_2 L_1)^* d = L_1^* L_2^* d.$$

Usage

```
sf_chain (oper1,oper2, adj,add, nm,nd,nt, mod,dat,tmp)
```

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. $m \leftarrow (L_2 L_1)^* d$ or $m \leftarrow m + (L_2 L_1)^* d$.

`add` addition flag (`bool`). If `true`, then $d \leftarrow d + (L_2 L_1) m$ or $m \leftarrow m + (L_2 L_1)^* d$ 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*`).

6.9.2 sf_cchain

The same as `sf_chain` but for complex data.

Usage

```
sf_cchain (oper1,oper2, adj,add, nm,nd,nt, mod, dat, tmp)
```

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. $m \leftarrow (L_2 L_1)^* d$ or $m \leftarrow m + (L_2 L_1)^* d$.
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d + (L_2 L_1) m$ or $m \leftarrow m + (L_2 L_1)^* d$ 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*`).

6.9.3 sf_array

For two operators L_1 and L_2, it calculates:

$$d = Lm,$$

or its adjoint

$$m = L^* d,$$

where

$$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \quad \text{and} \quad d = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

Usage

```
sf_array (oper1,oper2, adj,add, nm,nd1,nd2, mod,dat1,dat2)
```

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*`).

6.9.4 sf_normal

Applies a normal operator (self-adjoint) to the model, i.e. it calculates

$$d = LL^* m.$$

Usage

```
sf_normal (oper, add, nm,nd, mod,dat,tmp)
```

Input parameters

`oper`	the operator, L (`sf_operator`).
`add`	addition flag (`bool`). If `true`, then $d \leftarrow d + LL^* m$ is computed.
`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*`).

6.9.5 sf_chain3

Chains three operators L_1, L_2 and L_3:

$$d = (L_3 L_2 L_1) m.$$

Its adjoint is

$$m = (L_3 L_2 L_1)^* d = L_1^* L_2^* L_3^* d.$$

Usage

```
void sf_chain3 (oper1,oper2,oper3, adj,add, nm,nt1,nt2,nd, mod,dat,tmp1,tmp2)
```

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 $m \leftarrow m + L_1^* L_2^* L_3^* d$ is computed.

`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*`).

6.10 Dot product test for linear operators (dottest.c)

Performs the dot product test (see p. 118), to check whether the adjoint of the operator is coded incorrectly. Coding is incorrect if any of

$$\langle L m_1, d_2 \rangle = \langle m_1, L^* d_2 \rangle \quad \text{or} \quad \langle d_1 + L m_1, d_2 \rangle = \langle m_1, m_2 + L^* d_2 \rangle$$

does not hold (within machine precision). m_1 and d_2 are random vectors.

6.10.1 sf_dot_test

`dot1[0]` must equal `dot1[1]` and `dot2[0]` must equal `dot2[1]` for the test to pass.

Usage

```
sf_dot_test (oper, nm, nd, dot1, dot2)
```

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*`).