Elementary Science Operators

Many calculations we do in Geophysics, as well as in other disciplines of science and engineering, are linear operations and can be expressed as matrix multiplications. Many practitioners do not think of these calculations as linear operations or matrix multiplications, since these operators are usually derived and programmed as efficient special purpose programs, instead as a general matrix multiplication.

In this chapter, we show the object oriented implementation of some elementary operators and their adjoints. We start by discussing the obviously linear matrix multiplication operator and consequently look at increasingly more complex operators.

matrix multiply	conjugate-transpose matrix multiply
zero padding	truncation
causal integration	anticausal integration
convolution	crosscorrelation
stretching (moveout)	squeezing

MATRIX MULTIPLICATION

[1] The prototype of a linear operation is matrix multiplication. Multiplication of a matrix $\mathbf{B}$ times a vector $\mathbf{x}$ is computed by $y_i = \sum_j b_{ij} x_j$. The adjoint operation to the multiplication of the matrix $\mathbf{B}$ is multiplication with $\mathbf{B}$'s transpose: $\tilde{x}_j = \sum_i b_{ij} y_i$.

fopMatmul::apply

```
void fopMatmul::apply( floatArray & x, floatArray & y, int adj,
                       const Axislist & xal, const  Axislist & yal) const {
    int m = ::rows(x);
    int n = ::rows(matrix);
    for( int i=0; i<n; i++ )
    for( int j=0; j<m; j++ )
       if (adj)
          x(j) +=  matrix(i,j) * y(i);
       else
          y(i) +=  matrix(i,j) * x(j);
}
```

[1] Matthias Schwab

The CLOP library class `fopMatmul` is used to build matrix multiplication operators. The `fopMatmul` class contains the `apply` method which codes the mathematical relations concerning the forward and adjoint of matrix multiplication. Note the interchange of the input and output variables in the bottom `if` statement: this symmetry is characteristic for forward and adjoint operator pairs. In general, each input variable `x(i)` which contributes to a certain output variable `y(j)` in the forward operation will receive an equally weighted contribution from this variable `y(j)` in the adjoint operation.

In order to carry out a matrix multiplication, a user first creates a `fopMatmul` operator by invoking one of the two constructors listed in the `fopmatmul` header file. Having two constructors allows users to use matrix multiplication whether they have named coordinate axes and labels or simply a matrix and a vector.

fopMatmul::header

```
class fopMatmul : public fopone {
  public:
    fopMatmul( const floatspace &mat );
    fopMatmul( const Axis &inaxis, const Axis &outaxis, const floatArray &mat);
  }
```

After creation of a matrix multiplication operator, the application program can apply the operator to any conformable vector by invoking the `forward()` method of `fopMatmul`. The function `fopMatmul.forward()` accepts the input vector as argument and returns the multiplied vector. Analogously, the application program can apply the adjoint operator by invoking the `adjoint()` method.

CAUSAL INTEGRATION

[2] Causal integration is defined as

$$y(t) = \int_{-\infty}^{t} x(t)\; dt \tag{1}$$

Sampling the time axis gives a matrix equation which we should call causal summation, but we often call it causal integration.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} \tag{2}$$

[2]Hector Urdaneta

(In some applications the 1 on the diagonal is replaced by 1/2.) Causal integration is the simplest prototype of a recursive operator. The coding is trickier than the first derivative operator. Notice when you compute y_5 that it is the sum of 6 terms, but that this sum is more quickly computed as $y_5 = y_4 + x_5$. Thus equation (2) is more efficiently thought of as the recursion

$$y_t \quad = \quad y_{t-1} + x_t \qquad \text{for increasing } t \tag{3}$$

(which may also be regarded as a numerical representation of the differential equation $dy/dt = x$.)

When it comes time to think about the adjoint, however, it is easier to think of equation (2) than of (3). Let the matrix of equation (2) be called $\mathbf{C}$. Transposing to get $\mathbf{C}'$ and applying it to $\mathbf{y}$ gives us something back in the space of $\mathbf{x}$, namely $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$. From it we see that the adjoint calculation, if done recursively, needs to be done backwards like

$$\tilde{x}_{t-1} \quad = \quad \tilde{x}_t + y_{t-1} \qquad \text{for decreasing } t \tag{4}$$

We can sum up by saying that the adjoint of causal integration is anticausal integration.

The header `fopcausint.h` (page 3) defines the causal integrator constructor.

```
class fopcausint : public fopone {
  public:

     // construct
    fopcausint( const Axis & causintaxis );

  }
```

The code for anticausal integration is not obvious from the code for integration and the adjoint coding tricks we learned earlier. To understand the adjoint, you need to inspect the detailed form of the expression $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$ and take care to get the ends correct.

The class method `fopcausint::apply fopcausint` (page 3) shows the code for the forward and adjoint operation.

```
void fopcausint::apply(floatArray & x, floatArray & y, int adj,
                        const Axislist & xlist, const Axislist & ylist) const {

    int n = xlist.list[0].length;
    if (!adj) {
        y(0)   = x(0);
        for( int i=1;   i<n; i++) {   y(i) = y(i-1) + x(i); }
    } else {
        x(n-1) = y(n-1);
        for( int i=n-1; i>0; i--) { x(i-1) = x(i)   + y(i-1); }
    }
}
```

A beautiful example is shown in Figure 1.

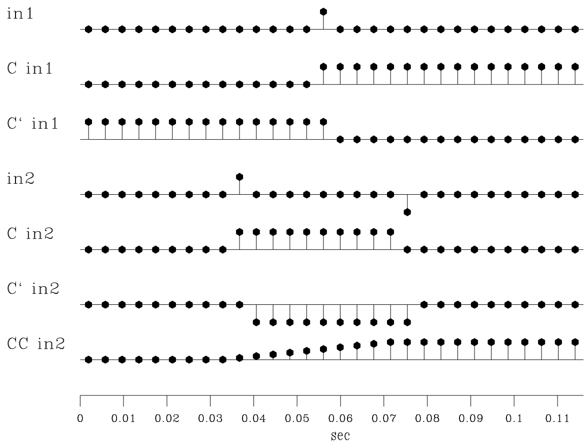

Figure 1: `in1` is an input pulse. `C in1` is its causal integral. `C' in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is the negative. `CC in2` is the double causal integral of `in2`. How can an equilateral triangle be built? elementary-cllp [NR]

FIRST DERIVATIVE

[3] Given a sampled signal, its time derivative can be estimated by convolution with the filter $(1, -1)/\Delta t$. This example arises so frequently that I display the matrix multiply below:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{5}$$

The filter impulse response is seen in any column in the middle of the matrix, namely $(1, -1)$. In the transposed matrix the filter impulse response is time reversed to $(-1, 1)$. So, mathematically, we can say that the adjoint of the time derivative operation is the negative time derivative. This corresponds also to the fact that the complex conjugate of $-i\omega$ is $i\omega$. We can also speak of the adjoint of the boundary conditions: we might say the adjoint of "no boundary condition" is "specified value" boundary conditions.

An example of the first derivative operator is shown in Figure 2.

The header file `fopdiff.h` (page 5) defines the `fopDiff()` class, where the first derivative operator is implemented. The class method `fopDiff::apply fopdiff` (page 5) performs the first derivative operation in either the forward or adjoint direction.

```
class fopDiff : public fopone {
  public:

      // construct
    fopDiff(const Axis& diffaxis );
```

[3] Hector Urdaneta

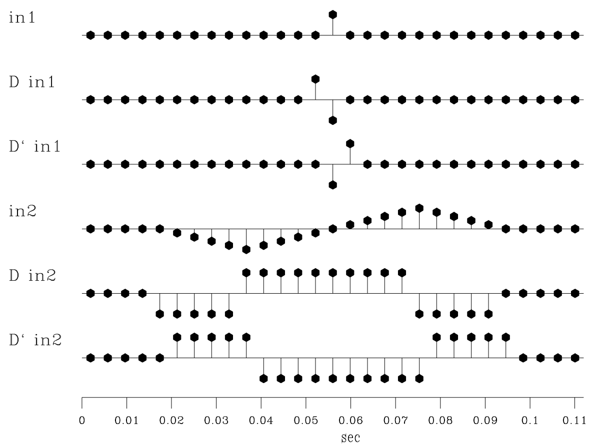

Figure 2: `in1` is an input pulse. `D in1` is its first derivative. `D' in1` is the adjoint of the first derivative operator applied of the pulse. `in2` is a double triangle function. Its first derivative `D in2` is made by three boxes and its adjoint `D' in2` is the negative. elementary-dllp [NR]

```
}


void fopDiff::apply(floatArray & x, floatArray & y, int adj,
                    const Axislist & xlist, const Axislist & ylist) const {

    for (int i=0; i<ylist.list[0].length; i++) {
        if (!adj) {
            y(i) = x(i+1) - x(i);
        } else {
            x(i+1) = x(i+1) + y(i);
            x(i  ) = x(i  ) - y(i);
        }
    }
}
```

CONVOLUTION

[4] In chapter ?? we discussed the implementation of convolution with transient end effects in great detail. However, we restricted the discussion to a straightforward convolution. In practise, filtering does often not consist of just convolving the data with a filter, but it involves shifting the data to some preferred time alignment, and truncating the output to the length of the input.

fopContrunc constructors

```
fopContrunc::fopContrunc( floatspace & x, int shift, int cut, int usedelta){
        Axislist ax = getinfo(x); num=3;
        ops    = new foperator*[num];
        ops[2] =                new fopContran( x,           usedelta ) ;
        ops[1] =                new fopAdvance( ax[0].name, shift    ) ;
        ops[0] = new fopAdjoint(new fopPad(      ax[0].name, 0, cut    ));
}
fopContrunc::fopContrunc(floatArray& x, int shift, int cut, const char* axname ){
        Axis ax(0,0,0,axname,1,1,1); num=3;
        ops    = new foperator*[num];
        ops[2] =                new fopContran( x, ax.name          ) ;
```

[4] Matthias Schwab

```
        ops[1] =                new fopAdvance(    ax.name, shift  ) ;
        ops[0] = new fopAdjoint(new fopPad(        ax.name, 0, cut ));
}
```

Chaining the convolution operator `fopContran` (see chapter ??) and the shift operator `fopAdvance` with the padding operator `fopPad` (see chapter ??), CLOP achieves the desired, more compliant convolution. The operator class `fopChain`, which implements such concatenations of existing CLOP operators, reuses the tested code of the composite operators `fopContran`, `fopAdvance`, and `fopPad`. Note, that the `fopPad` operator truncates a data set if the parameter `cut` is negative. The `forward()` method of the `fopChain` operator calls the `forward()` functions of its composite operators. The `adjoint()` operator of the `fopChain` operator is calling the adjoint functions of the chained operators, but in the reversed order.

fopContrunc header

```
class fopContrunc : public fopChain {
  public:
        fopContrunc( floatspace & x, int shift,int cut,int usedelta);
        fopContrunc( floatArray & x, int shift,int cut,const char* axname );
};
```

There exists a multitude of variations of convolution operators. One potentially interesting class for filter design and spectral estimation, are convolution schemes which do not assume zero data beyond their given data interval. One way to design such a filter is to truncate the output to an interval smaller than the input signal. Such a filter is easily implemented by changing the truncation operator in the operator chain described above. An alternative filter which avoids a zero assumption outside of the given input interval is a "circular" filter.

Figure 3 shows a simple test of different convolution operators and their end effects.

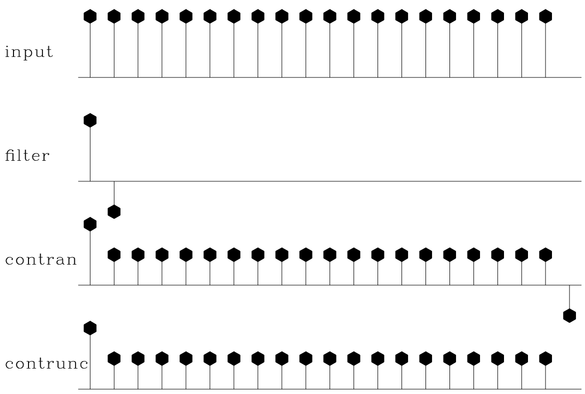

Figure 3: Demonstration of convolution. From top to bottom: (1) input; (2) filter; (3) output of `fopcontran()`; (4) output of `fopcontrunc()`; elementary-conv [NR]

A GENERAL MOVEOUT OPERATOR

[5] Many basic geophysical processes have at their core a squeezing/stretching operation. The familiar processes **migration**, **CDP stack**, **velocity analysis** and **slant stack** all utilize such an operator. In this chapter we lay the foundation and design a basic trace squeezing/stretching class, from which more complicated operators in chapter **??** and **??** are derived. We illuminate the workings of such a general operator by applying it to a simple process, namely normal moveout (NMO). For example NMO stretches traces of CDP gathers according to a given velocity. In 2-D, each trace in a time-offset panel is transformed into an equivalent moveout corrected trace.

The general moveout class `fopMO` accomplishes this task. It is based on the notion that it operates on a panel that is specified by two axes and a moveout function is defined that maps each input trace into an output trace. The input panel will have a time coordinate t and a location x, the output trace will have a time coordinate τ and a location x, and the mapping function will be of the form $t(x) = t(\tau, x)$.

The moveout transformation for one trace $\mathbf{N}$ is representable as a square matrix (when the time axes are of the same length). The matrix $\mathbf{N}$ is a (τ, t)-plane containing all zeros except an interpolation operator centered along the trajectory defined by the mapping function. The dots in the matrix below are zeros. The input signal d_t is put into the vector $\mathbf{d}$. The output vector for this particular operator $\mathbf{m}$—i.e., the moved out signal—is simply $(d_6, d_6, d_6, d_7, d_7, d_8, d_8, d_9, d_{10}, 0)$. In real life examples the subscript goes up to about one thousand instead of merely to ten.

$$
\mathbf{m} = \mathbf{N}\mathbf{d} = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \\ m_9 \\ m_{10} \end{bmatrix} = \begin{bmatrix}
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & . & 1 & . & . & . \\
. & . & . & . & . & . & 1 & . & . & . \\
. & . & . & . & . & . & . & 1 & . & . \\
. & . & . & . & . & . & . & 1 & . & . \\
. & . & . & . & . & . & . & . & 1 & . \\
. & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & .
\end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ d_{10} \end{bmatrix} \tag{6}
$$

You can think of the matrix as having a horizontal t-axis and a vertical τ-axis. The 1's in the matrix are arranged on the trajectory of the mapping function. $t = t(\tau)$.

The class definition for `fopMO` defines a constructor, an `apply()` function and two virtual functions, `mapping()` and `weight()`.

```
class fopMO : public fopone {
  public:

    // Constructors for fopMO: parameters are:
    // Time axis,Offset axis, slowness
    // or slowness array with as many samples as the time axis.
    fopMO(Axis & t ,Axis & x );
    fopMO(Axis & ti ,Axis & xi, Axis & to ,Axis & xo );

    ~fopMO();
```

[5] Martin Karrenbach

```
    // define moveout curve
    virtual float mapping( float tau, float x ) const ;

    // define weighting function for data points
    virtual float weight ( float tau, float x ) const ;

    void apply(floatArray &,floatArray &,int,
               const Axislist &,const Axislist &) const;

        Axislist inlist;
        Axislist outlist;
};
```

The heart of the stretching operation is implemented in `fopMO:apply()`. The code loops over the second input axis placing each sample in the input trace value into the appropriate output sample.

```
void fopMO::apply(floatArray & in,floatArray & out,int adj,
                  const Axislist & inlist,const Axislist & outlist) const {
    for( int  i=0;  i<inlist[1].length; i++)  { float x   = inlist[1].value(i);
    for( int iz=0; iz<inlist[0].length; iz++) { float tau = inlist[0].value(iz);

        int it = outlist[0].index( mapping(tau,x) );
        if (it < outlist[0].length && it >= 0) {
           if ( adj==0 ){ out(it,i) +=  in(iz,i) * weight(tau,x);} // modeling
           else         {  in(iz,i) += out(it,i) * weight(tau,x);} // processing
        }
    }}
}
```

The computation of the location is specified by a virtual member function `mapping()`. This function is redefined in a specific derived operator. The `mapping` function returns a floating point number which must be converted into an index in the input trace. This is achieved by using the `index()` member function of the `Axis` class (see man page **??**).

The mapping function

The basic stretching operator allows specification of a mapping function `fopMO::mapping()`. The mapping functions takes an input sample location (τ, x) as arguments and computes the output location in time (t) for the same x coordinate. Thus it is a one-dimensional mapping between input and output space. The mapping function is a virtual function, that can be overwritten by derived operators. In chapter **??** and **??** we extensively use this feature to specify a variety of mapping functions, but still rely on all other properties of the basic class.

```
float fopMO::mapping(float tau, float x) const {
   float  t = tau;
   return t;
}
```

The mapping function can be as complicated as you would like. By default, the `fopMO::mapping` is the identity matrix and will map a point in the input to the same location in the output, thus basically copying the input space onto the output space.

Weighting during the mapping

The basic stretching operator also allows specificaton of a weighting function `fopMO::weight()`. When the mapping is performed in the operator `fopMO` a weight can be multiplied onto the mapped data points. Weighting functions can be used to make an operator more unitary, or implement cut-off regions. If weights are applied the 1's in the matrix representation described above would become some time and offset dependent weight function. By default the `fopMO` operator will apply a weight of 1 to all data points (meaning essentially no weight is applied at all.)

```
float fopMO::weight(float tau, float x) const {
   float  dataweight = 1.;
   return dataweight;
}
```

Any derived class can override the virtual `weight()` function to implement its own desired weighting. Weighting functions can be used to specify data cut-offs like applying mute functions or aperture contraints to the data. Such applications are of considerable practical importance.

The fopMAP class

The class `fopMAP` (`fopmap.h` (page 9)) is derived from `fopMO`. It has the additional property that it stores internally some data required to calculate the mapping function. The data stored is an array of slowness values, one for each output time sample. The constructors for `fopMAP` allow the user to supply either a single slowness value or an array of different slowness values.

The `fopMAP` class does not define a mapping function, this means that it inherits `fopMO`'s mapping function which is usually redefined in the derived classes. Each class defines a mapping function that describes how to use the slowness data to calculate a coordinate transformation.

```
class fopMAP : public fopMO {
  public:
    fopMAP(Axis & t, Axis & x, float slow);
    fopMAP(Axis & t, Axis & x, float *slowarray);

    fopMAP(Axis & ti, Axis & xi, Axis & to, Axis & xo, float slow);
    fopMAP(Axis & ti, Axis & xi, Axis & to, Axis & xo, float *slowarray);

    float *slow;                /* mapping knows about slowness */

};
```

Many moveout operators can be derived from `fopMAP` using this interface. As a matter of fact, **all** operators in chapter **??** and **??** are derived from the basic class `fopMAP`. We do not show here an application of the basic operators `fopMO` and `fopMAP`, but rather point you to those chapters full of applications. In chapter **??** derived operators are described, that implement **normal moveout, linear moveout, kirchhoff moveout** and **anelliptic moveout**. Those operators are derived from `fopMAP` and merely specify their own new mapping function. Similarly the more comlicated composite operators, **velocity analysis**, **slant stack** and **Kirchhoff migration**, in chapter **??** are based on the derived operators introduced in chapter **??**. We have a good reason for having a description of such general operators in this chapter. In particular we want to show you how careful design of a few basic operators avoids code duplication and leads to a simplified implementation. Thus it creates a foundation that allows rapid design and testing of new operators and new interpolation schemes.