Base and Building Block Operators

[1] This chapter is an introduction to the basic framework and building blocks of operators. It serves as an overview for programmers intending to apply already existing operators or to combine them to form new operators. Using and understanding operators is discussed in chapter ??.

This document assumes a working knowledge of the C++ programming language, in particular the concepts of inheritance, class and construction.

BASIC OPERATOR HIERARCHY

Operators are arranged in the form of an inheritance hierarchy. This design eliminates the need to repeate code and simplifies the writing of new operators. At the top of the hierarchy are base classes which provide useful functions for derived operators and allow code to be written for abstract operators. New operators may be derived from any point in the hierarchy. Figure 1 depicts the dependencies between the operators discussed in this chapter.

For an introduction to operators and their functions, refer to the section on operators in chapter ??.

Base class foperator

At the top of the hierarchy is the `foperator`. It provides abstract virtual definitions of functions `Forward` and `Adjoint` operating from `floatspacearray` to `floatspacearray`. Derived classes must define these functions.

`foperator` provides functions which allow access to the `floatArray` and `Axislist` of a `floatspace`, and the `rows` and `cols` of a `floatspacearray`.

The `foperator` base class is also responsible for keeping track of the number of references to a specific `foperator`. This is hidden from the user for most applications and may usually be ignored. It is most important to note that `deref()` should be used for `foperator`s instead of `delete`. See man pages `operator` p. ??.

See `foperator.h` below.

```
class foperator {
 public:
  foperator() {references=1;}
```

[1]Lisa Laane

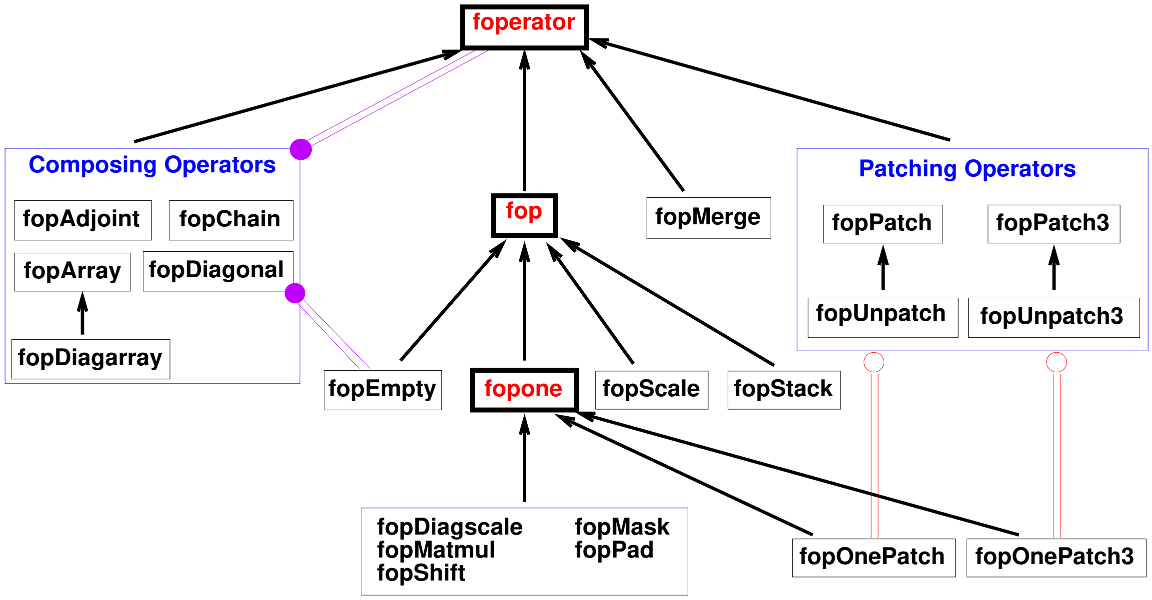

Figure 1: Graph of dependencies. Class relationships are depicted according to the Booch Method: a derived class points to its parent class with a single line and arrow, a hollow dot and double lines connect a class with the class it uses, and a solid dot and double lines connect a class with the class it contains. (basics-graph) [NR]

```
  virtual floatspacearray Forward(const floatspacearray &) const =0;
  virtual floatspacearray Adjoint(const floatspacearray &) const =0;

  virtual ~foperator() {references--;}

  void ref() {references++;}
  void deref() {references--;
                if (!(references)) delete this;}

  protected:
    int rows(const floatspacearray & x) const { return x.rows; }
    int cols(const floatspacearray & x) const { return x.cols; }

    floatArray & getdata(floatspace & s) const { return s.array; }
    const floatArray & getdata(const floatspace & s) const { return s.array; }
    Axislist & getinfo(floatspace & s) const { return s.axislist; }
    const Axislist & getinfo(const floatspace & s) const { return s.axislist; }
}
```

Base class fop

Directly derived from `foperator` is class `fop`. `fop` defines new virtual functions for `Forward` and `Adjoint`, this time from a single `floatspace` to a single `floatspace`. It serves as the base class for operators which operate between `floatspaces`. Any derived class must define `Forward` and `Adjoint` between spaces, as well as a constructor.

`Forward` and `Adjoint` from `floatspacearray` to `floatspacearray` are defined to cycle through the `floatspaces` of the input `floatspacearray` and return the result of applying the operator to each `floatspace` individually. See man pages `op` p. ??.

Many simple operators are derived from `fop`, but `fopone` (described in the next section) provides additional helper functions which usually make it the preferred class for derivation.

See `fop.h` below.

```
class fop : public foperator {

  public:
    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;

    virtual floatspace Forward(const floatspace &) const = 0;
    virtual floatspace Adjoint(const floatspace &) const = 0;

};
```

Base class fopone

The majority of operators are derived from `fopone`, which is derived from `fop`. `fopone` includes several functions to help handle the work shared by most operators. When called `Forward` or `Adjoint`, it creates the output space, arranges the input space (through transposes) so that the axes are in the right order for calculating, and cycles over any extraneous axes. For derived operators, the user does not rewrite the `Forward` and `Adjoint` operators. Instead, an `apply()` function is written that is passed the input and output arrays. For more detail on the functions used by `fopone` see man pages `opone` p. ?? and chapter ?? on using and understanding an operator. See `fopone.h` below.

```
class fopone : public fop {
  public:
    fopone();

    // Construct fopone given input Axislist and
    // output Axislist.
    fopone( const Axislist &, const Axislist & );

    floatspace Forward( const floatspace & ) const;
    floatspace Adjoint( const floatspace & ) const;

    floatspacearray Forward( const floatspacearray & x ) const
      { return fop::Forward( x ); }
    floatspacearray Adjoint( const floatspacearray & x ) const
      { return fop::Adjoint( x ); }

    virtual Axislist forwardinfo( const Axislist & ) const;
    virtual Axislist adjointinfo( const Axislist & ) const;

  }
```

COMPOSING OPERATORS

Several operators derived from `foperator` are available to compose other operators and apply them from `floatspacearray` to `floatspacearray`. Using composing operators allows users to combine and easily create more powerful and complex operators.

Since the composing operators are `foperator`s themselves, they may be used in any combination with themselves and each other to create arrays of arrays, adjoints of chains, arrays of chains of chains, and so on.

It should be noted that the composing operators can assign an operator to be contained from either an operator or a pointer to one. In most cases, the operator to be contained can simply be passed, but when it may go out of scope *before* the composing operator, it should be constructed by calling `new` on a pointer to it, and the *pointer* should be passed. This case is likely to arise when composing operators are constructed within other operators and not in the main routine. See the man pages for `opAdjoint` p. **??**, `fopChain` p. **??**, and `fopArray` p. **??** for more information. See also the introduction to chains, adjoints and arrays in chapter **??**.

The Adjoint operator

The simplest composing operator is `fopAdjoint`. When applied `Forward`, it will call the `Adjoint` of the `foperator` from which it is built. When applied `Adjoint`, it will perform the `Forward` of the original `foperator`.

The constructor takes the operator or a pointer to the operator of which the adjoint operator is desired. `fopAdjoint` can also be created as follows:

```
fopAdjoint A = !B;
```

where **A** is the the adjoint of `foperator` **B**. See `fopadjoint.h` below.

```
class fopAdjoint : public foperator {
  public:
    fopAdjoint( foperator &);
    fopAdjoint( foperator *);

    fopAdjoint & operator=(const fopAdjoint &);

    friend fopAdjoint operator!(foperator &);

    ~fopAdjoint();

    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;
  }
```

Chains of operators

`fopChain` is used to join two `foperators` to be applied in sequence to a `floatspacearray`. When applied `Forward`, `fopChain` applies the second `foperator` and then the first. When applied `Adjoint`, the operators are applied in the opposite order and adjoint. Derived classes may contain more than two `foperators` in the chain.

Chains are constructed from the two `foperators` or pointers to them. Chains can also be created by using the following syntax:

```
fopChain C = A * B;
```

where **A** and **B** are `foperator`s and **C** is a chain of **A** acting on **B**. When using this syntax, any number of `foperators` may be joined using `*`.

See `fopchain.h` below.

```
class fopChain : public foperator {
 public:
    fopChain() {};
    fopChain(const fopChain &);
    fopChain(foperator &,foperator &);
    fopChain(foperator *, foperator *);

    fopChain & operator=(const fopChain &);
    friend fopChain operator*(foperator &, foperator &);

    ~fopChain();

    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;
  }
```

Arrays of operators

`fopArray` is used to create two-dimensional arrays of `foperator`s to be applied to `floatspacearray`s (which are also up to two-dimensional) in a fashion resembling matrix multiply. For `Forward`, the rows of the `fopArray` are multiplied by the columns of the `floatspacearray`, and this is summed along. `Adjoint` does the same, except the transpose of the `fopArray` is used.

`fopArray`s are constructed by specifying the number of rows and columns in the array. An individual `foperator` can then be assigned with the `set` function. `set` takes the coordinates of the position in the array and either a `foperator` or a pointer to one. Every position in the array must be assigned an operator. Individual operators may be viewed or applied (but not changed) using an indexing operator.

See `foparray.h` below.

```
class fopArray : public foperator {
  public:
    fopArray() {rows=0; cols=0; ops=0;}
    fopArray(const fopArray &);
    fopArray & operator=(const fopArray &);
    ~fopArray();

    fopArray(int i, int j=1);

    // index a specific operator; cannot change return value
    const foperator & operator() (int i, int j=1) const;

    // set to operator
    void set(int, int, foperator &);
    void set(int, int, foperator *);

    void setrow( int, foperator &);
    void setrow( int, foperator *);

    void setcol( int, foperator &);
    void setcol( int, foperator *);

    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;
```

```
}
```

Diagonal arrays of operators

For convenience, class `fopDiagarray` is provided to create diagonal arrays of `foperators`.

Derived from `fopArray` it provides the same functions as its parent class. It is constructed from the length of a side of the square array and a `foperator` to be assigned to the diagonal positions. All other positions are set to the "zero" operator, `fopEmpty` (see the discussion of `fopEmpty` in Section). Any position of the array can be reassigned to another `foperator`.

One should note that this is *not* an optimized version of `fopArray`, and only serves as an aid to easily set the diagonal positions to the same operator and all other positions to "zero". See the following section on `fopDiagonal`. See `fopdiagarray.h` below.

```
class fopDiagarray : public fopArray {
 public:
  fopDiagarray(int, foperator &);
  fopDiagarray(int, foperator *);
};
```

Optimized diagonal arrays of operators

The operator `fopDiagonal` is the optimized version of the diagonal array of operators. It behaves like a `fopArray`, but is restricted to contain non-"zero" operators *only* along the diagonal.

`fopDiagonal` operators can be constructed from the number of rows (or columns) of the array, or that number and an operator to insert in every position in the diagonal. The class provides functions to index and set the diagonal positions of the array by specifying the row number (which is the same as the column number). See `fopdiagonal.h` below.

```
class fopDiagonal : public foperator {
  public:
    fopDiagonal( int );
    fopDiagonal( int, foperator & op );
    fopDiagonal( int, foperator * op );
    fopDiagonal( const fopDiagonal &);

    fopDiagonal & operator=(const fopDiagonal &);
    ~fopDiagonal();

    void set( int, foperator & op );
    void set( int, foperator * op );
    const foperator & operator()( int ) const;

    floatspacearray Forward( const floatspacearray & ) const;
    floatspacearray Adjoint( const floatspacearray & ) const;
  }
```

UTILITY OPERATORS

The following is a list of simple operators that are basic to many applications and can easily be used by programmers.

More complete descriptions are provided in the chapter ?? Appendix man pages.

Operators derived from foperator

Any operator derived from `foperator` defines `Forward` and `Adjoint` from `floatspacearray` to `floatspacearray`.

```
floatspacearray Forward( const & floatspacearray ) const
floatspacearray Adjoint( const & floatspacearray ) const
```

The merge operator

The `Forward` operator merges a vector of `floatspaces` (in the form of a `floatspacearray`) into a `floatspace` (returned as a 1x1 `floatspacearray`) that has one more dimension than the individual spaces in the input. Note that the individual spaces *must* have the same dimensions. The `Adjoint` operator slices a `floatspacearray` containing a single `floatspace` at every sample along the given `Axis` to form a vector `floatspacearray` of spaces with one fewer dimension than the input space.

The constructor takes an `Axis` that describes the dimension to be created or eliminated. See `fopmerge.h` below.

```
class fopMerge : public foperator {

    public:
        fopMerge(const Axis &);

        floatspacearray Forward(const floatspacearray &) const;
        floatspacearray Adjoint(const floatspacearray &) const;

    }
```

Figure 2 shows a `floatspacearray` of four `floatspaces` (each of two dimensions) being merged into a single three-dimensional `floatspace`.

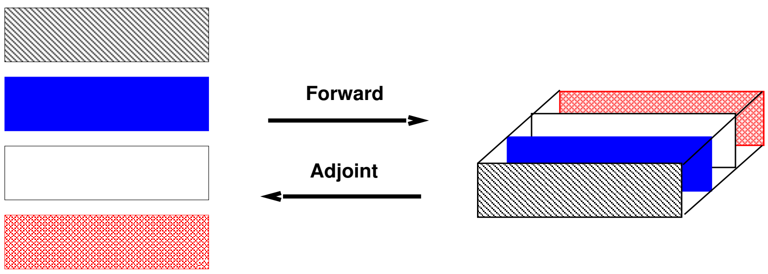

Figure 2: Left frame = input floatspacearray of 2-D spaces; Right frame = merged 3-D floatspace (basics-merge) [NR]

The patch operator

The operator `fopPatch` can chop a space into subspaces (which may overlap) called patches. The `Adjoint` adds them back again. Because of possible overlap, however, weighting functions are needed to restore the original space (see the following section on the unpatch operator).

`Forward` takes a 1x1 `floatspacearray` and returns a one-dimensional `floatspacearray` of patches. `Adjoint` takes a one-dimensional `floatspacearray` of patches and returns a 1x1 `floatspacearray` of the patches put together into a single space. `fopPatch` uses `fopOnepatch` for each individual patch that is extracted from or inserted in the larger space. See the description of `fopOnepatch` in Section .

`fopPatch` is constructed from the number and size of patches along each `Axis`, and the `Axislist` of the single large space in the intput `floatspacearray`. See `foppatch.h` below.

`fopPatch` operates on `floatspacearray`s containing two-dimensional `floatspace`s and patches. `fopPatch3` is the equivalent operator for three-dimensional patching. See `foppatch3.h` below.

```
class fopPatch : public foperator {
    friend class fopUnpatch;
  public:
    fopPatch();
    fopPatch(const fopPatch &);
    fopPatch( int num0,int num1,int size0,int size1,const Axislist& );

    floatspacearray Forward( const floatspacearray & ) const;
    floatspacearray Adjoint( const floatspacearray & ) const;

  }
```

```
class fopPatch3 : public foperator {
    friend class fopUnpatch3;
  public:
    fopPatch3();
    fopPatch3(const fopPatch3 &);
    fopPatch3( int, int, int, int, int, int, const Axislist & );

    floatspacearray Forward( const floatspacearray & ) const;
    floatspacearray Adjoint( const floatspacearray & ) const;

  }
```

The unpatch operator

Since patches may overlap, the `Adjoint` of `fopPatch` is not the same as the inverse. `fopUnpatch` is derived from `fopPatch` to serve as the inverse. It uses weighting functions so that the `Adjoint` of `fopUnpatch` is the inverse of `Forward` of `fopPatch`, and vice versa. Note that the `Adjoint` is the preferred method for reassembling patches.

`fopUnpatch` is based on the following equation:

$$\tilde{\mathbf{d}} = [\mathbf{W}_{wall}\ \mathbf{P}'\ \mathbf{W}_{wind}\ \mathbf{P}\ ]\mathbf{d} \tag{1}$$

where $\mathbf{d}$ is the original data, $\tilde{\mathbf{d}}$ is the reconstructed data, $\mathbf{P}$ is the `Forward` of `fopPatch`, and $\mathbf{P}'$ is the `Adjoint` of `fopPatch`. $\mathbf{W}_{wind}$ is an arbitrary weighting function for a window (patch). $\mathbf{W}_{wall}$ is the weighting function for the wall (full space) that is calculated from $\mathbf{W}_{wind}$ to undo the effects of overlapping. Since $\mathbf{W}_{wall}$ is constructed from $\mathbf{W}_{wind}$, *any* weighting function may be chosen for $\mathbf{W}_{wind}$.

`Adjoint` for `fopUnpatch` applies the following to a `floatspacearray` of patches:

$$\mathbf{W}_{wall}\ \mathbf{P}'\ \mathbf{W}_{wind}$$

and `Forward` applies the following to a `floatspacearray` containing a single space:

$$\mathbf{W}_{wind}\ \mathbf{P}\ \mathbf{W}_{wall}$$

`fopUnpatch` may be constructed the same way as `fopPatch`. It may also be constructed from an object of type `fopPatch`. See `fopunpatch.h` below.

`fopUnpatch3` is the equivalent operator for three-dimensional unpatching. See `fopunpatch3.h` below.

```
class fopUnpatch : public fopPatch {
  public:
    fopUnpatch( const fopPatch & );
    fopUnpatch( int, int, int, int, const Axislist & );
    floatspacearray Adjoint( const floatspacearray & ) const;
    floatspacearray Forward( const floatspacearray & ) const;
 }


class fopUnpatch3 : public fopPatch3 {
  public:
    fopUnpatch3( const fopPatch3 & );
    fopUnpatch3( int, int, int, int, int, int, const Axislist & );
    floatspacearray Adjoint( const floatspacearray & ) const;
    floatspacearray Forward( const floatspacearray & ) const;
  }
```

The following exmple creates and applies an operator, **op**. It first makes a 9x1 `floatspacearray` of patches of size 4x7 each. It then applies a 9x9 diagonal array of scaling operators to scale the 4th patch by 3.0 and all others by 0.5. Finally, it reassembles the patches (with weighting).

```
fopPatch myPatch( 3, 3, 4, 7, mySpaceAxisList );
fopScale myScale(0.5);
fopScale myScale2(3.0);
fopDiagonal myDiagonal( 3*3 , myScale );
myDiagonal.set( 4, myScale2 );
fopUnpatch myUnpatch( myPatch );

fopChain op = !myUnpatch * myDiagonal * myPatch;

floatspacearray result = op.Forward( mySpacearray );
```

Figure 3 depicts the single `floatspace` in the `floatspacearray` before and after application of the operator.

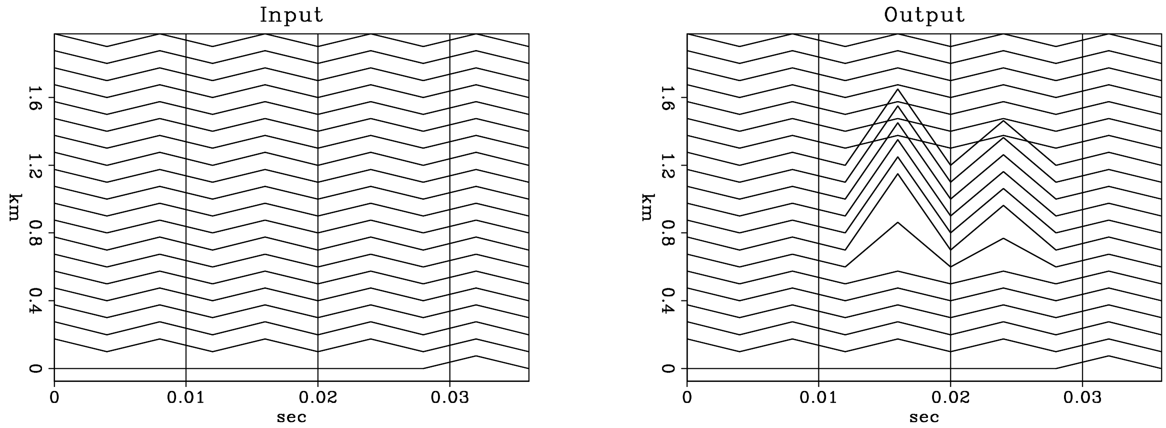

Figure 3: Left frame = input floatspace, right frame = output floatspace basics-patchout [NR]

Operators derived from fop

`Forward` and `Adjoint` for operators derived from `fop` operate between single `floatspace`s. When called on a `floatspacearray` they cycle through all the `floatspace`s of the array.

```
floatspace Forward( const & floatspace ) const
floatspace Adjoint( const & floatspace ) const

floatspacearray Forward( const & floatspacearray ) const
floatspacearray Adjoint( const & floatspacearray ) const
```

The scale operator

`fopScale` multiplies a `floatspace` by a single float.

It is constructed from the float by which the space is to be multiplied. `Forward` multiplies by the float, and `Adjoint` also multiplies by the complex conjugate of the float. See `fopscale.h` below.

```
class fopScale : public fop {
  public:
    // Create a scale operator that multiplies by the argument
    fopScale(float);

    floatspace Forward(const floatspace &) const;
    floatspace Adjoint(const floatspace &) const;

    floatspacearray Forward(const floatspacearray & x) const
      { return fop::Forward(x); }
    floatspacearray Adjoint(const floatspacearray & x) const
      { return fop::Adjoint(x); }

  }
```

The "empty" operator

`fopEmpty` serves as a "zero" operator. When applied to a `floatspace` either `Forward` or `Adjoint`, it returns a "zero" `floatspace`, which acts like a "zero" when added to, subtracted from, or multiplied by another `floatspace`. See man pages `space` p. ?? and `Axis` p. ??.

`fopEmpty` has no constructor and is created by a simple declaration. See `fopempty.h` below.

```
class fopEmpty : public fop {
 public:
    floatspace Forward(const floatspace &) const;
    floatspace Adjoint(const floatspace &) const;

    floatspacearray Forward(const floatspacearray & x) const
      { return fop::Forward(x); }
    floatspacearray Adjoint(const floatspacearray & x) const
      { return fop::Adjoint(x); }

};
```

The stack operator

`fopStack` applies a reduction or expansion along one axis. When applied `Forward`, it adds an extra dimension to the space, and the data is replicated across the extra dimension. When applied `Adjoint`, it removes that dimension from the space by summing over the extra dimension.

The constructor is given the axis to be stacked or spread over and optionally the position at which it is inserted. If no position is specified the forward operator inserts the extra axis as the first axis of the space. See `fopstack.h` below.

```
class fopStack : public fop {
  public:
    // supply axis to be stacked over
    // and optionally the position to insert it
    fopStack(const Axis & inax , int pos = -1 );

    floatspacearray Forward(const floatspacearray & x) const;
    floatspacearray Adjoint(const floatspacearray & x) const;

  }
```

Figure 4 shows a two-dimensional `floatspace` being spread into a three-dimensional `floatspace` in the `Forward` direction. The `Adjoint` operator sums over the third axis to create a two-dimensional `floatspace`.

Operators derived from fopone

Since `fopone` is derived from `fop`, `fopone` operators can apply `Forward` and `Adjoint` from `floatspace` to `floatspace` or from `floatspacearray` to `floatspacearray`. If applied to a `floatspacearray`, they will apply the operator to each space in the array seperately.

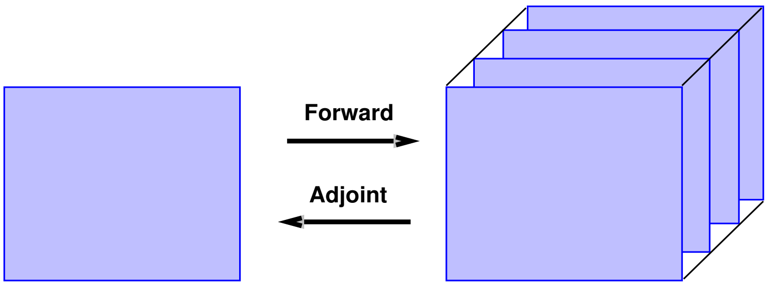

Figure 4: Left frame = input 2-D floatspace, right frame = replicated 3-D floatspace (basics-stack) [NR]

The diagonal scale operator

`fopDiagscale` scales each element of a `floatspace` by a constant.

It is constructed either from a single `floatspace` of constants or two `floatspaces`, `sp1` and `sp2` of constants. In the second case, the input `floatspace` is multiplied by the square root of `sp1/sp2`. See `fopdiagscale.h` below.

```
class fopDiagScale : public fopone {
  public:
    // Construct a fopDiagScale that scales each
    // element of a space by a constant.
    fopDiagScale( const floatspace & scalars );
    fopDiagScale( floatspace & sp1, floatspace & sp2 );

  }
```

The mask operator

`fopMask` applies a mask to a `floatspace`, meaning it sets some of the elements to zero.

The constructors take the axis to mask along and the positions along this axis to set to zero. `Forward` and `Adjoint` perform the same function. See `fopmask.h` below.

```
class fopMask : public fopone {
  public:
    // Construct from Axis to apply to,
    // and an array of int, used as a mask.
    fopMask(const Axis &,const int *);

    // Construct from the Axis and a floatspace used
    // to mask.
    fopMask(const Axis &,const floatspace & x);
    ~fopMask();
  }
```

The one patch operator

`fopOnepatch` creates a single two-dimensional patch from a two-dimensional `floatspace` when applied `Forward`. When applied `Adjoint` it puts a patch in a larger `floatspace` that contains zeros elsewhere.

`fopOnepatch3` does the same for three dimensions.

One patches are constructed in many ways, always with specifications for the full space first and a patch second. The specific constructors are described in `foponepatch` and `foponepatch3`.

```
class fopOnepatch : public fopone {
  public:
    // construct from axislist of the full space and patch origin
    // and length along first axis and second axis
    fopOnepatch( const Axislist & spacelist,
                 float origin0, int len0,
                 float origin1, int len1 );

    // construct from origin and
    // length along each axis of full space
    // and axislist of the patch
    fopOnepatch( float origin0, int len0, float origin1, int len1,
                 const Axislist & patchlist );

    // construct from full space axislist and patch corner coordinates
    fopOnepatch( const Axislist & spacelist,
                 float i0, float j0, float i1, float j1);

    // construct from full space and patch axislists
    fopOnepatch( const Axislist & spacelist, const Axislist & patchlist );

  }


class fopOnepatch3 : public fopone {
  public:
    // Construct from axislist of the  full  space  and  patch
    // origin and length along each axis.
    fopOnepatch3( const Axislist & spacelist,
                  float origin0, int len0,
                  float origin1, int len1,
                  float origin2, int len2 );

    // Construct from full space origin and length
    // along each axis
    // and axislist of the patch
    fopOnepatch3( float origin0, int len0,
                  float origin1, int len1,
                  float origin2, int len2,
                  const Axislist & patchlist );

    // construct from full space axislist and patch corner coordinates
    fopOnepatch3( const Axislist & spacelist,
                  float i0, float j0, float k0,
                  float i1, float j1, float k1 );

    // construct from full space and patch axislists
    fopOnepatch3( const Axislist& spacelist,const Axislist& patchlist );

  }
```

Figure 5 shows the application of `fopOnepatch` `Forward` and `Adjoint` to first make a patch from the original `floatspace` and then put the patch back in a `floatspace` of the original size.

See the discussion of patch and unpatch in Section .

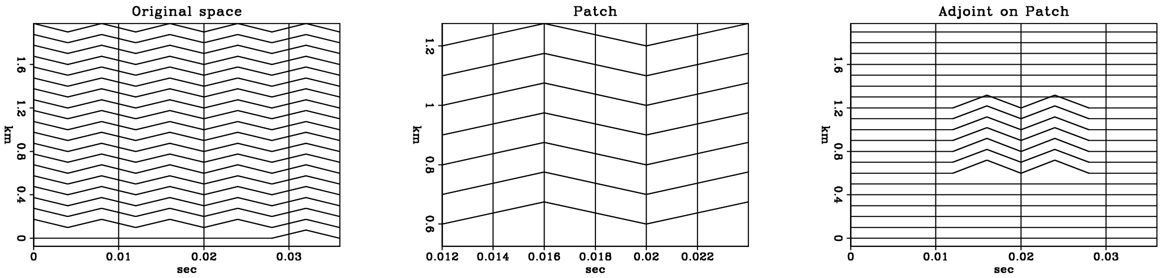

Figure 5: Left frame = original space, middle frame = fopOnepatch applied Forward on the original space. A single patch of size 5x6, right frame = fopOnepatch applied Adjoint on the patch basics-onepatchout [NR]

The pad operator

`fopPad` pads or truncates along an `Axis` of the `floatspace`. Surrounding a dataset by zeros (zero padding) is adjoint to throwing away the extended data (truncation). Let us see why this is so. Set a signal in a vector $\mathbf{x}$, and then make a longer vector $\mathbf{y}$ by adding some zeros at the end of $\mathbf{x}$. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \left[\begin{array}{c} \mathbf{I} \\ \mathbf{0} \end{array} \right] \; \mathbf{x} \tag{2}$$

The matrix is simply an identity matrix $\mathbf{I}$ above a zero matrix $\mathbf{0}$. To find the transpose to zero padding, we now transpose the matrix and do another matrix multiply:

$$\tilde{\mathbf{x}} \quad = \quad \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \end{array} \right] \; \mathbf{y} \tag{3}$$

So the transpose operation to zero padding data is simply *truncating* the data back to its original length.

The constructor for `fopPad` may either take the `Axis` to pad or its name, followed by the number of samples to pad to the front and to the back. For `Forward`, positive numbers specify the number to pad and negative numbers the number to truncate. `Adjoint` does the opposite.

See `foppad.h` below.

```
class fopPad : public fopone {
  public:
    fopPad(const Axis &, int, int);
    fopPad(const char *, int, int);

  }
```

The shift operator

`fopShift` shifts data in the `floatspace` along an `Axis`.

The constructors take either the `Axis` or the name of the `Axis` to shift along, and the distance by which to shift (positive values advance and negative values retard the origin). See `fopshift.h` below.

```
class fopShift : public fopone {
  public:
    fopShift(const Axis & inaxis,int shift);
    fopShift(const char* axname,int shift);
  }
```