Stanford Exploration Project, Report SERGEY, May 1, 2000, pages 1–619

Short Note

Simple linear operators in Fortran 90

Sergey Fomel and Jon Claerbout[1]

INTRODUCTION

A linear operator maps an input vector to an output vector. In the adjoint mode, the mapping direction is reverse. The simplest implementation of this idea is a minimal interface `operator( adj, add, model, data)`, where the logical variable `adj` defines adjoint or forward mode, and variable `add` defines whether the output of the program should be added to the previous value of the corresponding actual argument. The minimal interface is the "mathematical" connection to operators as *objects*. To provide the "geophysical" connection, we need to initialize an operator with the arguments that *construct* it. Unlike object-oriented languages, such as C++, Fortran 90 doesn't have a built-in mechanism for constructing the objects. It provides, however, a secure way for encapsulating the construction variables in modules. This paper shows some examples of translating Fortran 77 operator *subroutines* to Fortran 90 operator *modules*. In all the examples, both the model and the data vectors are represented in the minimal interface as one-dimensional real-number arrays. In geophysical data processing, such arrays can correspond to data traces, sampled in time. The array data type is naturally embedded in Fortran 90, which allows us to use a lot of intrinsic constructs. However, the general idea of minimal interfaces applies equally well to more complicated data structures. The first natural use of the minimal interface is a generic dot-product routine, testing whether an adjoint operator complies with its formal definition. The second use, not covered in this paper, is a generic framework for estimating the solutions of inverse problems with methods of mathematical optimization.

UNIVERSAL DOT-PRODUCT ROUTINE

Dot-product routine is a numerical implementation of the definition of adjoint operator. We use it to confirm that the dot product of vectors $\mathbf{y}$ and $\mathbf{Ax}$ is equal (up to the machine precision) to the dot product of $\mathbf{x}$ and $\mathbf{A}'\mathbf{y}$ for random $\mathbf{x}$ and $\mathbf{y}$. The dot-product test is crucially important for applications, where the composite operators $\mathbf{A}'\mathbf{A}$ or $\mathbf{AA}'$ need to be positive definite. An important example is the conjugate gradient optimization. As a rule, an unusually high misfit in the dot-product test indicates either a conceptual error or a programming bug. The dot-product test is an example of a mathematical abstraction, which requires object-oriented

[1]**email:** sergey@sep.stanford.edu, jon@sep.stanford.edu

programming style to be expressed adequately. With Fortran 77 tools, we have to write a separate dot-product test program for each operator. Each time the dot-product test fails, we need to check both the operator implementation, and the implementation of the test itself. Ideally, the dot-product test routine should be isolated from the operator and precompiled in a library. The concept of function interface, introduced in Fortran 90, allows us to achieve this goal in a straightforward way. Module `dottest` is the actual implementation of a universal routine. An operator, passed to the `dot_test` subroutine is defined by its interface

```
function oper (adj, add, mod, dat) result (stat)
   integer              :: stat
   logical, intent (in) :: adj, add
   real, dimension (:)  :: mod, dat
end function oper
```

When the logical argument `adj` is `.false.`, the operator maps the model vector `mod` (defined as a one-dimensional real array[2]) to the data vector `dat`. When `adj=.true.`, `dat` is mapped into `mod`. When `add` is `.true.`, the output is added to the previous value of the corresponding vector.

```
module dottest
  use random_mod

contains

  subroutine dot_test (oper, n_mod, n_dat, dotprod, dotadd)
    integer,             intent (in)  :: n_mod, n_dat
    real, dimension (2), intent (out) :: dotprod, dotadd
    interface
       function oper (adj, add, mod, dat) result (stat)
         integer              :: stat
         logical, intent (in) :: adj, add
         real, dimension (:)  :: mod, dat
       end function oper
    end interface
    optional                          :: dotadd

    real, dimension (n_mod)           :: mod1, mod2
    real, dimension (n_dat)           :: dat1, dat2
    integer                           :: stat
    logical, parameter                :: T = .true., F = .false.

    call random (mod1) ; call random (dat2)

    stat = oper (F, F, mod1, dat1) ; dotprod (1) = dot_product (dat1, dat2)
    stat = oper (T, F, mod2, dat2) ; dotprod (2) = dot_product (mod1, mod2)

    write (0,*) dotprod ; if (.not. present (dotadd)) return
```

[2]Geophysical data that we routinely process in a computer can always be represented as a set of real numbers, packed into a one-dimensional array. On the other hand, the mathematical concept of a vector can handle much more abstract constructions than simple arrays. As a language for engineering problems, Fortran 90 doesn't handle abstractions as well as other advanced languages such as C++. In particular, there is no elegant way to represent the idea of an abstract vector or an abstract vector space. However, the operator abstraction, presented in this paper, can be repeated to any concrete data type that we may find deserving special treatment.

```
    stat = oper (F, T, mod1, dat1) ; dotadd  (1) = dot_product (dat1, dat2)
    stat = oper (T, T, mod2, dat2) ; dotadd  (2) = dot_product (mod1, mod2)

    write (0,*) dotadd
  end subroutine dot_test

end module dottest
```

Before the `dot_test` program calls a particular operator, we need to supply the operator with the additional parameters it needs for execution. In order to do that, we encapsulate the operator function in one module with the initialization routine. The following section provides examples of modules for simple basic linear operators.

EXAMPLES OF LINEAR OPERATORS

Once the linear operator interface is defined, it leads to a precise formal definition for canonical linear operator function. According to that definition, a canonical operator should be provided with a standard interface and should pass the generic dot-product test. Simple linear operators, discussed in this paper, form the foundation of a canonical library of linear operators, implemented in Fortran 90.

Matrix multiply

If both the input and the output of a linear operator are vectors, composed of real numbers, the operator is equivalent to multiplication by a real matrix. In data processing problems, we hardly ever see this matrix. We prefer instead to represent it in a compact non-matrix form and to write a subroutine that performs the forward operator and its adjoint. Nevertheless, multiplication by a matrix is a very important example of a linear operator, serving as the basic prototype for many other operators. Module `matmult` implements both the operation $y_i = \sum_j b_{ij} x_j$ to multiply a vector $\mathbf{x}$ by a matrix $\mathbf{B}$ and the operation $\tilde{x}_j = \sum_i b_{ij} y_i$ to multiply the data vector $\mathbf{y}$ by the transpose matrix $\mathbf{B}'$. The matrix multiplication is preceded by a call to subroutine `adjnull`, defined in module `adj`, which optionally erases the output according to the value of the logical parameter `add`. When `add=.true.`, the result of applying the operator is accumulated in the output, e.g. `y=y+B*x`.

```
module adj_mod
contains

  subroutine adjnull (adj, add, x, y)

    logical, intent (in) :: adj, add
    real, dimension (:)  :: x, y

    if (.not. add) then
       if (adj) then
          x = 0.
       else
          y = 0.
       end if
    end if

  end subroutine adjnull
```

```
end module adj_mod


module matmult
  use adj_mod

  real, dimension (:,:), pointer, private :: b

contains

  subroutine matmult_init (matrix)
    real, dimension(:,:), pointer :: matrix
    b => matrix
  end subroutine matmult_init

  function matmult_op (adj, add, x, y) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: x, y

    integer              :: i

    call adjnull (adj, add, x, y)

    do i = 1, size (x)
       if (adj) then
          x (i) = x (i) + sum (b (:, i) * y (:))
       else
          y (:) = y (:) +      b (:, i) * x (i)
       end if
    end do

    stat = 0
  end function matmult_op

end module matmult
```

Subroutine `matmult_init` serves as the object constructor, copying a pointer to the matrix in a *private* variable `b`, local with respect to the module, but accessible by the actual operator function `matmult_op`. The operator function returns an integer `stat`. In our convention, the value `stat=0` corresponds to a normal execution of the operator routine. Otherwise, it can return the diagnostic of an error. Fortran 90 has intrinsic functions `matmul` and `dot_product` for matrix-vector manipulations, which we could have used in this simple example. We prefer more explicit style to show how the algebraic symmetry of mathematical constructions translate into actual code. We will see more examples of this symmetry in the following examples.

Transient convolution

As with any linear operator, the convolution operator has a representation in the form of matrix multiplication. However, it is more convenient to implement it in a different form. Compare the mathematical definition of convolution $y_k = \sum_i b_i x_{k-i}$ with its implementation in module `tcai1`. The name of this module abbreviates *Transient Convolution, Adjoint is the Input 1-D*, which refers to the fact that the adjoint operator cross-correlates the output with the filter to return to the input space: $x_j = \sum_i b_i y_{i+j}$.

```
module tcai1
  use adj_mod

  real, dimension (:), pointer, private :: b
  integer, private                      :: nb, nx

contains

  subroutine tcai1_init (filter, nd)
    real, dimension (:), pointer :: filter
    integer, intent (in)         :: nd

    b => filter
    nb = size (b)
    nx = nd
  end subroutine tcai1_init

  function tcai1_op (adj, add, x, y) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: x, y

    integer              :: i

    call adjnull (adj, add, x, y)

    do i = 1, nb
       if (adj) then
          x (1 :     nx    ) = x (1 :     nx    ) + b (i) * y (i : i + nx - 1)
       else
          y (i : i + nx - 1) = y (i : i + nx - 1) + b (i) * x (1 :     nx    )
       end if
    end do

    stat = 0

  end function tcai1_op

end module tcai1
```

Analogously to the case of matrix multiplication, we initialize the operator with a pointer to the filter. The pointer is stored in a private variable `b` to be referenced by function `tcai1_op`. In some applications, we need to consider the filter **b** as the output of the adjoint process. In this case, the adjoint operation corresponds to cross-correlation of a fixed portion of the filter input **x** with a variable portion the output **y**: $b_i = \sum_j x_j y_{i+j}$. Module `tcaf1` (*Transient Convolution, Adjoint is the Filter 1-D*) does the job.

```
module tcaf1
  use adj_mod

  real,    dimension (:), pointer, private :: x
  logical, dimension (:), pointer, private :: m
  integer, private                         :: nx

contains
```

```
  subroutine tcaf1_init (input)
    real,    dimension (:), pointer :: input

    x => input ; nx = size (x)
  end subroutine tcaf1_init

  function tcaf1_op (adj, add, b, y) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: b, y

    integer              :: i

    call adjnull (adj, add, b, y)

    do i = 1, size (b)
       if (adj) then
          b (i) = b (i) + sum (y (i : i + nx - 1) * x (1 : nx))
       else
          y (i : i + nx - 1) = y (i : i + nx - 1) + x (1 : nx) * b (i)
       end if
    end do

    stat = 0
  end function tcaf1_op

end module tcaf1
```

Internal convolution

The output of the transient convolution program has the length larger than the length of the input. In many applications, it is convenient to set the two lengths equal to each other. Reformulating the convolution operator allows us to relate the output space to the input space and deal appropriately with the convolution end effects. The Fortran 90 implementation is in module `icai1` (*Internal Convolution, Adjoint is the Input 1-D*).

```
module icai1
  use adj_mod

  real, dimension (:), pointer, private :: b
  integer,                      private :: lg, nb, nx, y1, y2

contains

  subroutine icai1_init (filter, nd, lag)
    real, dimension (:), pointer   :: filter
    integer,           intent (in) :: nd
    integer, optional, intent (in) :: lag

    integer                        :: na

    b => filter ; nb = size (b) ; nx = nd
```

```
    if (present (lag)) then
       lg = lag
    else
       lg = 1
    end if

    y1 = nb + 1 - lg
    y2 = nx + 1 - lg
  end subroutine icai1_init

  function icai1_op (adj, add, x, y) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: x, y

    integer              :: i, x1, x2

    call adjnull (adj, add, x, y)

    do i = 1, nb
       x1 = nb + 1 - i
       x2 = nx + 1 - i
       if (adj) then
          x (x1 : x2) = x (x1 : x2) + b (i) * y (y1 : y2)
       else
          y (y1 : y2) = y (y1 : y2) + b (i) * x (x1 : x2)
       end if
    end do

    stat = 0
  end function icai1_op

end module icai1
```

The lag parameter `lag`, saved at the initialization step to the private variable `lg`, defines how the output of the operator aligns with the input. The end effects of transient and internal convolutions are compared in Figure 1.

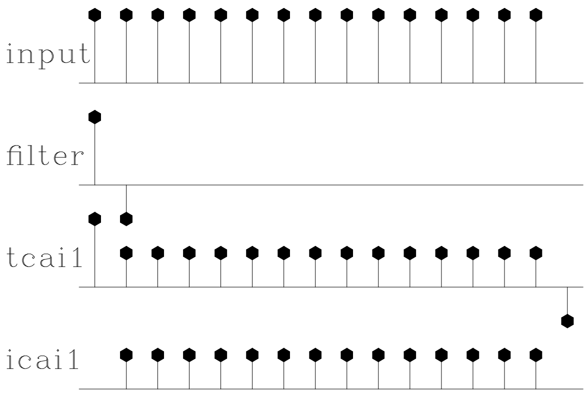

Figure 1: Example of convolution end effects. From top to bottom: (1) input; (2) filter; (3) output of `tcai1()`, (4) output of `icai1()` with (`lag=1`). gee90-conv90 [ER]

Causal and leaky integration

Causal integration in the continuous world is defined as

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

A more general operator is leaky integration, defined as

$$y(t) \;=\; \int_{0}^{\infty} x(t-\tau)\,e^{-\alpha\tau}\,d\tau\;. \tag{2}$$

As $\alpha \to 0$, leaky integration becomes causal integration. An efficient numerical implementation of a simple causal integrator employs the fact that operator (1) solves the differential equation

$$dy/dt = x(t)\,, \tag{3}$$

and operator (2) solves

$$dy/dt = \alpha\,y(t) + x(t)\;. \tag{4}$$

In the discrete world, equation (4) leads to the recursion scheme

$$y_t \;=\; \rho\,y_{t-1} + x_t \qquad t \text{ increasing}\,, \tag{5}$$

where $\rho = 1$ for $\alpha = 0$. The adjoint of (5) is the recursion

$$\tilde{x}_{t-1} \;=\; \rho\tilde{x}_t + y_{t-1} \qquad t \text{ decreasing}\,, \tag{6}$$

which corresponds to anticausal leaky integration

$$\tilde{x}(t) \;=\; \int_{0}^{\infty} y(\tau-t)\,e^{-\alpha\tau}\,d\tau\;. \tag{7}$$

Module `leakint` implements causal and anticausal integration.

```
module leakint
  use adj_mod

  real, private :: r = 1.

contains

  subroutine leakint_init (rho)
    real, intent (in) :: rho
    r = rho
  end subroutine leakint_init

  function leakint_op (adj, add, x, y) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: x, y

    call adjnull (adj, add, x, y)

    if (adj) then
       call integrate (y,size (y),1,x)
    else
```

```
      call integrate (x,1,size (x),y)
   end if

   stat = 0

 contains

   subroutine integrate (inp, nb, ne, out)
     integer,             intent (in)  :: nb, ne
     real, dimension (:), intent (in)  :: inp
     real, dimension (:), intent (out) :: out

     integer                           :: i
     real                              :: t

     t = 0.
     do i = nb, ne, sign (1, ne - nb)
        t = r * t + inp (i)
        out (i) = out (i) + t
     end do
   end subroutine integrate

 end function leakint_op

end module leakint
```

The operator function `leakint_op` contains and calls the helper subroutine `integrate`, which implements recusions (5) and (6). Figure 2 illustrates the program for $\rho = 1$.

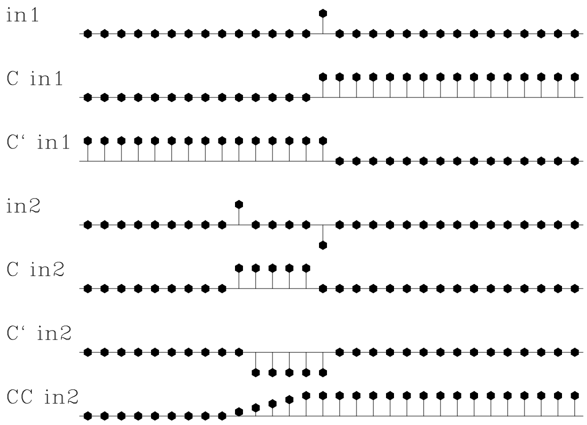

Figure 2: `in1` is an input pulse. `C in1` is its causal integral. `C′ in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is the negative. `CC in2` is the double causal integral of `in2`. gee90-causint90 [ER]

Nearest-neighbor interpolation

Binning irregularly sampled data to a regular grid is another important operator, often found in practical applications. The adjoint of binning is interpolation, and the simplest case of interpolation is *nearest-neighbor interpolation*, where each data point is assigned with the nearest grid point value. Irregular data typically come in ordinate-coordinate pairs. According to the minimal interface design, coordinates as well as the grid information belong to the initialization interface, while the corresponding ordinates go directly to the operator interface. At the initialization step, module `bin1` transforms the grid and coordinate information into two data-size arrays: the integer array x with the nearest bin locations and the logical array m, which masks the data points outside of the grid. Additionally, there is a real array w with the weighting function, optionally supplied by the user.

```
module bin1
  use adj_mod

  integer,                               private :: nd
  integer, dimension (:), allocatable, private :: x
  real,    dimension (:), allocatable, private :: w
  logical, dimension (:), allocatable, private :: m

contains

  subroutine bin1_init (coord, o1, d1, n1, weight)
    real, dimension (:), intent (in) :: coord, weight
    real,                intent (in) :: o1, d1
    integer,             intent (in) :: n1
    optional                         :: weight

    nd = size (coord)
    if (.not. allocated (x)) allocate (x (nd), m (nd), w (nd))

    x = 1.5 + (coord - o1) / d1
    m = (x < 1 .or. x > n1)

    if (present (weight)) then
       m = m .or. weight == 0.
       where (.not. m) w = weight
    else
       where (.not. m) w = 1.
    end if
  end subroutine bin1_init

  function bin1_op (adj, add, mod, ord) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: mod, ord

    integer              :: i

    call adjnull (adj, add, mod, ord)

    do i = 1, nd ; if (m (i)) cycle
       if (adj) then
          mod (x (i)) = mod (x (i)) + w (i) * ord (   i )
       else
          ord (   i ) = ord (   i ) + w (i) * mod (x (i))
       end if
    end do

    stat = 0
  end function bin1_op

  subroutine bin1_close ()
    deallocate (x, m, w)
```

```
  end subroutine bin1_close

end module bin1
```

The output of the interpolation operator is "smooth", each data point obtains a unique value from the nearest bin. In a shorthand Fortran 90 notation, we could write the forward operation in one line as `where (.not. m) ord = ord + w * mod (x)`. However, writing the adjoint operation as `where (.not. m) mod (x) = mod (x) + w * ord` is not correct, because the output of the adjoint process (binning) is "rough": each bin may receive zero, one, or more than one input from the data.

Linear interpolation

A more accurate interpolation method is *linear interpolation*. To interpolate linearly, we locate each data coordinate between two model points on the grid. The data ordinate is defined by drawing a straight line through the model values or, in other words, by weighting them proportionally to the distance from the coordinate. The adjoint uses the same weights to distribute the data ordinate between the two grid points. The interfaces in module `lint1` are exactly the same as those in `bin1`, but the actual implementation details are slightly different. Using the similar interfaces makes these modules replaceable; in any application, one of them can be replaced by the other with a simple change of the `use` statement. The nearest neighbor interpolation is faster, while the linear interpolation provides a more accurate result.

```
module lint1
  use adj_mod

  integer,                             private :: nd
  integer, dimension (:), allocatable, private :: x
  real,    dimension (:), allocatable, private :: w1, w2
  logical, dimension (:), allocatable, private :: m

contains

  subroutine lint1_init (coord, o1, d1, n1, weight)
    real, dimension (:), intent (in) :: coord, weight
    real,                intent (in) :: o1, d1
    integer,             intent (in) :: n1
    optional                         :: weight

    nd = size (coord)
    if (.not. allocated (x)) allocate (x (nd), m (nd), w1 (nd), w2 (nd))

    w1 = (coord - o1) / d1
    x = w1     ; w2  = w1 - x
    x = x  + 1 ; w1 = 1. -  w2

    m = (x < 1 .or. x >= n1)
    if (present (weight)) then
        m = m .or. (weight == 0.)
       where (.not. m)
          w1 = w1 * weight ; w2 = w2 * weight
       end where
    end if
  end subroutine lint1_init
```

```
  function lint1_op (adj, add, mod, ord) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: mod, ord

    integer              :: i, i1, i2

    call adjnull (adj, add, mod, ord)

    do i = 1, nd ; if (m (i)) cycle
       i1 = x (i) ; i2 = i1 + 1
       if (adj) then
          mod (i1) = mod (i1) + w1 (i) * ord (i )
          mod (i2) = mod (i2) + w2 (i) * ord (i )
       else
          ord (i ) = ord (i ) + w1 (i) * mod (i1) + &
                                w2 (i) * mod (i2)
       end if
    end do

    stat = 0
  end function lint1_op

  subroutine lint1_close ()
    deallocate (x, m, w1, w2)
  end subroutine lint1_close

end module lint1
```

CONCLUSIONS

We have shown how to define simple linear *operators* as simple *objects* with initialization attributes and generic operator interfaces. The tools we have used for this task are Fortran 90 *modules* and function *interfaces*. The next step is to combine elementary operators into chains and blocks, creating more sophisticated composite operators, and to pass those operators to universal linear *estimators* for estimating the solutions of geophysical inverse problems.