Appendix A

RATional FORtran == Ratfor

Bare-bones **Fortran** is our most universal computer language for computational physics. For general programming, however, it has been surpassed by **C**. "**Ratfor**" is Fortran with C-like syntax. I believe Ratfor is the best available expository language for mathematical algorithms. Ratfor was invented by the people who invented C. Ratfor programs are converted to Fortran with the Ratfor **preprocessor**. Since the preprocessor is publicly available, Ratfor is practically as universal as Fortran.[1]

You will not really need the Ratfor preprocessor or any precise definitions if you already know Fortran or almost any other computer language, because then the Ratfor language will be easy to understand. Statements on a line may be separated by ";". Statements may be grouped together with braces { }. Do loops do not require statement numbers because { } defines the range. Given that `if( )` is true, the statements in the following { } are done. `else{ }` does what you expect. We may *not* contract `else if` to `elseif`. We may always omit the braces { } when they contain only one statement. `break` will cause premature termination of the enclosing { }. `break 2` escapes from {{ }}. `while( ) { }` repeats the statements in { } while the condition ( ) is true. `repeat { ... } until( )` is a loop that tests at the bottom. A looping statement more general than `do` is `for(initialize; condition; reinitialize) { }`. An example of one equivalent to `do i=0,n-1` is the looping statement `for(i=0;i<n;i=i+i)`. The statement `next` causes skipping to the end of any loop and a retrial of the test condition. `next` is rarely used, but when it is, we must beware of an inconsistency between Fortran and C-language. Where Ratfor uses `next`, the C-language uses `continue` (which in Ratfor and Fortran is merely a place holder for labels). The Fortran relational operators `.gt.`, `.ge.`, `.ne.`, etc. may be written >, >=, !=, etc. The logical operators `.and.` and `.or.` may be written & and |. Anything from a # to the end of the line is a comment. Anything that does not make sense to the Ratfor preprocessor, such as Fortran input-output, is passed through without change. (Ratfor has a `switch` statement but we never use it because

[1] **Kernighan, B.W.** and Plauger, P.J., 1976, Software Tools: Addison-Wesley. Ratfor was invented at AT&T, which makes it available directly or through many computer vendors. The original Ratfor transforms Ratfor code to **Fortran** 66. See `http://sepwww.stanford.edu/sep/prof` for a public-domain Ratfor translator to **Fortran** 77.

it conflicts with the `implicit undefined` declaration. Anybody want to help us fix the `switch` in public domain Ratfor?)

Indentation in `Ratfor` is used for readability. It is not part of the `Ratfor` language. Choose your own style. I have overcondensed. There are two **pitfall**s associated with indentation. The beginner's pitfall is to assume that a `do` loop ends where the indentation ends. The loop ends after the first statement. A larger scope for the `do` loop is made by enclosing multiple statements in braces. The other pitfall arises in any construction like `if() ... if() ... else`. The `else` goes with the last `if()` regardless of indentation. If you want the `else` with the earlier `if()`, you must use braces like `if() { if() ... } else ...`.

The most serious limitation of **Fortran**-77 is its lack of ability to allocate temporary memory. I have written a **preprocessor** to **Ratfor** or **Fortran** to overcome this memory-allocation limitation. This program, named **sat**, allows subroutines to include the declaration `temporary real data(n1,n2)`, so that memory is allocated during execution of the subroutine where the declaration is written. Fortran-77 forces us to accomplish something like this only with predetermined constants or parameters. If the **sat** preprocessor is not available on your system, you can modify the subroutines in this book by putting the appropriate numerical constants into the memory arrays being allocated, or adapt the programs here to Fortran 90 (although students at Stanford seem to prefer the **sat** approach).

Appendix B

Seplib and SEP software

At the time of writing, this book can be run on a variety of computers. You will have noticed that each figure caption contains a box enclosing a label. In the electronic book, this box is a pushbutton that generally activates a rebuilding of the figure, sometimes after program or parameter changes and sometimes interactively. The label in the box points to the location of the underlying software. My associates and I have worked through complete cycles of "**burn**ing" and building all the figures on various computers. To enable you to do the same, and to further enable you to rapidly build on my work, I have released electronic copies of my books.[1] This short appendix describes the utility software that is used extensively in the electronic book.

Most of the seismic utility software at **SEP**[2] (**SEP**) handles seismic data as a rectangular lattice or "cube" of numbers. Each cube-processing program appends to the history file for the cube. Preprocessors extend **Fortran** (or **Ratfor**) to enable it to allocate memory at run time, to facilitate input and output of data cubes, and to facilitate self-documenting programs.

At the **SEP** a library of subroutines known as **seplib** evolved for routine operations. These subroutines mostly handle data in the form of cubes, planes, and vectors. A cube is defined by 14 parameters with standard names and two files: one the data cube itself, and the other containing the 14 parameters and a history of the life of the cube as it passed through a sequence of cube-processing programs. Most of these cube-processing programs have been written by researchers, but several nonscientific cube programs have become highly developed and are widely shared. Altogether there are (1) a library of subroutines, (2) a library of main programs, (3) some naming conventions, and (4) a graphics library called **vplot**. The subroutine library has good manual pages. The main programs rarely have manual pages, their documentation being supplied by the on-line self-documentation that is extracted from the comments

[1] My previous book, "Earth soundings analysis: processing versus inversion," is available on CD-ROM titled **SEP**-CD-4 from Stanford University Press, telephone 415-723-1593 for $15. My current book, "Basic Earth Imaging" (BEI) has been released in incomplete form to sponsors of the **Stanford Exploration Project (SEP)** on disk SEP-CD-3.

[2] Old reports of the Stanford Exploration Project can be found in the library of the Society of Exploration Geophysicists in Tulsa, Oklahoma.

at the beginning of the source file. Following is a list of the names of popular main programs:

`Byte`	Scale floats to brightness bytes for raster display.
`Cat`	Concatenate conforming cubes along the 3-axis.
`Contour`	Contour plot a plane.
`Cp`	Copy a cube.
`Dd`	Convert between ASCI, floats, complex, bytes, etc.
`Dots`	Plot a plane of floats.
`Ft3d`	Do three-dimensional Fourier transform.
`Graph`	Plot a line of floats.
`In`	Check the validity of a data cube.
`Merge`	Merge conforming cubes side by side on any axis.
`Movie`	View a cube with Rick Ottolini's cube viewer.
`Noise`	Add noise to data.
`Reverse`	Reverse a cube axis.
`Spike`	Make a plane wave of synthetic data.
`Ta2vplot`	Convert a byte format to raster display with `vplot`.
`Tpow`	Scale data by a power of time t (1-axis).
`Thplot`	Make a hidden line plot.
`Transpose`	Transpose cube axes.
`Tube`	View a `vplot` file on a screen.
`Wiggle`	Plot a plane of floats as "wiggle traces."
`Window`	Find a subcube by truncation or subsampling.

To use the cube-processing programs, read this document, and then for each command, read its on-line self-documentation. To write cube-processing programs, read the manual page for `seplib` and the subroutines mentioned there and here. To write `vplot` programs, see the references on `vplot`.

B.1 THE DATA CUBE

The data cube itself is like a Fortran three-dimensional matrix. Its location in the computer file system is denoted by `in=PATHNAME`, where `in=` is the literal occurrence of those three characters, and `PATHNAME` is a directory tree location like `/sep/professor/pvi/data/western73.F`. Like the Fortran cube, the data cube can be real, complex, double precision, or byte, and these cases are distinguished by the element size in bytes. Thus the history file contains one of `esize=4`, `esize=8`, or `esize=1`, respectively. Embedded blanks around the "=" are always forbidden. The cube values are binary information; they cannot be printed or edited (without the intervention of something like a Fortran "format"). To read and write cubes, see the manual pages for such routines as `reed, sreed, rite, srite, snap`.

A cube has three axes. The number of points on the 1-axis is `n1`. A Fortran declaration of a cube could be `real mydata(n1,n2,n3)`. For a plane, `n3=1`, and for

a line, n2=1. In addition, many programs take "1" as the default for an undefined value of n2 or n3. The physical location of the single data value mydata(1,1,1), like a mathematical origin (o_1, o_2, o_3), is denoted by the three real variables o1, o2, and o3. The data-cube values are presumed to be uniformly spaced along these axes like the mathematical increments $(\Delta_1, \Delta_2, \Delta_3)$, which may be negative and are denoted by the three real variables d1, d2, and d3.

Each axis has a label, and naturally these labels are called label1, label2, and label3. Examples of labels are kilometers, sec, Hz, and "offset, km". Most often, label1="time, sec". Altogether that is $2 + 3 \times 4$ parameters, and there is an optional title parameter that is interpreted by most of the plot programs. An example is title="Yilmaz and Cumro Canada profile 25". We reserve the names n4,o4,d4, and label4 (a few programs support them already), and please do not use n5 etc. for anything but a five-dimensional cubic lattice.

B.2 THE HISTORY FILE

The 15 parameters above, and many more parameters defined by authors of cube-processing programs, are part of the "**history file**" (which is ASCI, so we can print it). A great many cube-processing programs are simple filters—i.e., one cube goes in and one cube comes out—and that is the case I will describe in detail here. For other cases, such as where two go in and one comes out, or none go in and one comes out (synthetic data), or one goes in and none come out (plotting program), I refer you to the manual pages, particularly to subroutine names beginning with aux (as in "auxiliary").

Let us dissect an example of a simple cube-processing program and its use. Suppose we have a seismogram in a data cube and we want only the first 500 points on it, i.e., the first 500 points on the 1-axis. A utility cube filter named Window will do the job. Our command line looks like

```
< mygiven.H Window n1=500 > myshort.H
```

On this command line, mygiven.H is the name of the history file of the data we are given, and myshort.H is the history file we will create. The moment Window, or any other seplib program, begins, it copies mygiven.H to myshort.H; from then on, information can only be appended to myshort.H. When Window learns that we want the 1-axis on our output cube to be 500, it does call putch('n1','i',500), which appends n1=500 to myshort.H. But before this, some other things happen. First, seplib's internals will get our log-in name, the date, the name of the computer we are using, and Window's name (which is Window), and append these to myshort.H. The internals will scan mygiven.H for in=somewhere to find the input data cube itself, and will then figure out where we want to keep the output cube. Seplib will guess that someone named professor wants to keep his data cube at some place like /scr/professor/_Window.H@. You should read the manual page for datapath to see how you can set up the default location for your datasets. The reason datapath

exists is to facilitate isolating data from text, which is usually helpful for archiving.

When a cube-processing filter wonders what the value is of `n1` for the cube coming in, it makes a subroutine call like `call hetch("n1","i",n1)`. The value returned for `n1` will be the *last* value of `n1` found on the history file. `Window` also needs to find a different `n1`, the one we put on the command line. For this it will invoke something like `call getch("n1","i",n1out)`. Then, so the next user will know how big the output cube is, it will `call putch("n1","i",n1out)`. For more details, see the manual pages.

If we want to take input parameters from a file instead of from the command line, we type something like `<in.H Window par=myparfile.p > out.H`. The `.p` is my naming convention and is wholly optional, as is the `.H` notation for a history file.

`Sepcube` programs are self-documenting. When you type the name of the program with no input cube and no command-line arguments, you should see the self-documentation (which comes from the initial comment lines in the program).

SEP software supports "pipelining." For example, we can slice a plane out of a data cube, make a contour plot, and display the plot, all with the command line

```
<in.H Window n3=1 | Contour | Tube
```

where, as in UNIX pipes, the "`|`" denotes the passage of information from one program to the next. Pipelining is a convenience for the user because it saves defining a location for necessary intermediate files. The history files do flow down UNIX pipes. You may not have noticed that some location had to be assigned to the data at the intermediate stages, and when you typed the pipeline above, you were spared that clutter. To write `seplib` programs that allow pipelining, you need to read the manual page on `hclose()` to keep the history file from intermingling with the data cube itself.

A sample history file follows: this was an old one, so I removed a few anachronisms manually.

```
 # Texaco Subduction Trench: read from tape by Bill Harlan
n1=1900 n2=2274
o1=2.4 it0=600 d1=.004 d2=50.  in=/d5/alaska

Window:   bill   Wed Apr 13 14:27:57 1983
input() :    in ="/d5/alaska"
output() : sets next in="/q2/data/Dalw"
Input:  float Fortran (1900,2274,1)
Output: float Fortran (512,128,1)
 n1=512 n2=128 n3=1

Swab:   root@mazama   Mon Feb 17 03:23:08 1986
# input history file /r3/q2/data/Halw
input() :    in ="/q2/data/Dalw"
output() : sets next in="/q2/data/Dalw_002870_Rcp"
#ibs=8192 #obs=8192

Rcp: paul Mon Feb 17 03:23:15 PST 1986
Copying from mazama:/r3/q2/data/Halw
```

```
to hanauma:/q2/data/Halw
in="/q2/data/Dalw"

Cp:   jon@hanauma   Wed Apr  3 23:18:13 1991
input() :    in ="/q2/data/Dalw"
output() : sets next in="/scr/jon/_junk.H@"
```

B.3 MEMORY ALLOCATION

`Sepcube` programs can be written in Fortran, Ratfor, or C. A serious problem with **Fortran**-77 (and hence Ratfor) is that memory cannot be allocated for arrays whose size is determined at run time. We have worked around this limitation by using two home-grown preprocessors, one called **saw** (Stanford Auto Writer) for main programs, and one called **sat** (Stanford Auto Temporaries) for subroutines. Both preprocessors transform either Fortran or Ratfor.

B.3.1 Memory allocation in subroutines with sat

The **sat** preprocessor allows us to declare temporary arrays of arbitrary dimension, such as

```
temporary real*4 data(n1,n2,n3), convolution(j+k-1)
```

These declarations must follow other declarations and precede the executable statements.

B.3.2 The main program environment with saw

The **saw** preprocessor also calls an essential initialization routine `initpar()`, organizes the self-doc, and simplifies data-cube input. See the on-line self-documentation or the manual pages for full details. Following is a complete **saw** program for a simple task:

```
# <in.H  Scale scaleval=1. > out.H
#
# Copy input to output and scale by scaleval
# keyword generic scale
#%
integer n1, n2, n3, esize
from history: integer n1, n2, n3, esize
if (esize !=4) call erexit('esize != 4')
allocate: real x(n1,n2)

subroutine scaleit( n1,n2, x)
integer i1,i2, n1,n2
```

```
real x(n1,n2), scaleval
from par: real scaleval=1.
call hclose() # no more parameter handling.
call sreed('in', x, 4*n1*n2)
do i1=1,n1
do i2=1,n2
x(i1,i2) = x(i1,i2) * scaleval
call srite( 'out', x, 4*n1*n2)
return; end
```

REFERENCES

Claerbout, J., 1990, Introduction to `seplib` and SEP utility software: **SEP–70**, 413–436.

Claerbout, J., 1986, A canonical program library: **SEP–50**, 281–290.

Cole, S., and Dellinger, J., Vplot: **SEP**'s plot language: SEP-**60**, 349–389.

Dellinger, J., 1989, Why does **SEP** still use Vplot?: SEP–**61**, 327–335.

B.4 BASIC SUBROUTINES

The subroutines `null()` (page 8), `copy()` (page 8), and `scale()` (page 8) used in this book are pedestrian.

```
subroutine null(     xx, n)
integer i, n;   real xx( n)
do i= 1, n
        xx(i) = 0.
return; end


subroutine scale( factor, n, data)
integer i,                n
real factor, data(n)
do i= 1, n
        data(i) = factor * data(i)
return; end


subroutine copy( n, xx, yy)
integer i, n;   real xx(n), yy(n)
do i= 1, n
        yy(i) = xx(i)
return; end
```

Appendix C

Entrance examination

1. (10 minutes) Given a residual $\mathbf{r}$ where

$$\mathbf{r} \quad = \quad \mathbf{d}_0 - m_1\mathbf{b}_1 - m_2\mathbf{b}_2 - m_3\mathbf{b}_3$$

   The data is $\mathbf{d}_0$. The fitting functions are the column vectors $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$ and the model parameters are the scalars m_1, m_2, and m_3. Suppose m_1 and m_2 are already known. Derive a formula for finding m_3 which minimizes the residual length (squared) $\mathbf{r} \cdot \mathbf{r}$.

2. (10 minutes) Below is a subroutine written in a mysterious dialect of Fortran. Describe ALL the inputs required for this subroutine to multiply a vector times the *transpose* of a matrix.

```
# matrix multiply and its adjoint
#
subroutine matmult( adj, bb,        x,nx,  y,ny)
integer ix, iy,     adj,              nx,    ny
real                      bb(ny,nx), x(nx), y(ny)
if( adj == 0 )
        do iy= 1, ny
                y(iy) = 0.
else
        do ix= 1, nx
                x(ix) = 0.
do ix= 1, nx {
do iy= 1, ny {
        if( adj == 0 )
                        y(iy) = y(iy) + bb(iy,ix) * x(ix)
        else
                        x(ix) = x(ix) + bb(iy,ix) * y(iy)
        }}
return; end
```