Chapter 11

RATional FORtran == Ratfor90

"**Ratfor**" is a dialect of Fortran that is more concise than raw Fortran. This book now runs on original `ratfor`, `ratfor77` and `ratfor90`. During 1997 I plan to take advantage of new features of `Fortran 90` and the book will become incompatible with earlier Ratfors.

The newest Ratfor "compiler," `ratfor90`, is a simple word-processing program (written[1] in Perl and freely distributed) that inputs an attractive Fortran-like dialect and outputs Fortran90. Mainly, the word-processor produces Fortran statements like `end do`, `end if`, `end program` and `end module`, from the Ratfor "}". Ratfor source is about 25-30% smaller than the equivalent Fortran, so it is equivalently more readable.

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. Ratfor was invented by the people[2] who invented C. After inventing C, they realized that they had made a mistake (too many semicolons) and they fixed it in Ratfor, although it was too late for C. Otherwise, Ratfor uses C-like syntax, the syntax that is also found in the popular languages **C++** and **Java**.

At SEP we supplemented Ratfor77 by preprocessors to give Fortran77 the ability to allocate memory on the fly. These abilities are built into Fortran90 and are seamlessly included in **Ratfor90**. To take advantage of Fortran90's new features while maintaining the concise coding style provided by Ratfor, we had to write a new Ratfor preprocessor, Ratfor90, which produces Fortran90 rather than Fortran77 code.

You should be able to read Ratfor if you already know Fortran or any similar computer language. Writing Ratfor is easy if you already know Fortran because written Fortran is valid Ratfor. You can mix Ratfor and Fortran. The Ratfor processor is not a compiler but a simple word-processing program which passes Fortran (which it does not understand) through unchanged. The Ratfor processor converts the Ratfor dialect to Fortran. To maximize the amount of Ratfor, you will need to know its

[1] The Ratfor90 preprocessor was written by my colleague, Bob Clapp.

[2] Kernighan, B.W. and Plauger, P.J., 1976, Software Tools: Addison-Wesley.

rules. Here they are:

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 omit the braces { } where they contain only one statement. `break` (equivalent to the Fortran90 `exit`) causes premature termination of the enclosing { }. `while( ) { }` repeats the statements in { } while the condition ( ) is true. Ratfor recognizes `repeat { ... } until( )` as a loop that tests at the bottom. `next` causes skipping to the end of any loop and a retrial of the test condition. `next` (equivalent to the Fortran90 `cycle` statement) is rarely used, and the Ratfor90 coder may write either `next` or `cycle`. Here we encounter 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. A line may be continued in Ratfor by ending it with the underscore charactor "_" (like Fortran90's &).

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 actually 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 ...`. Ratfor also recognizes the looping statement used in C, C++, and Java. It is `for(`initialize`;` condition`;` reinitialize`)` { }.

11.0.1 Changes and backward compatibility

We were forced to make one change to Ratfor90 because of new things in Fortran90. Ratfor77 allows & and | for the logical operators && and ||. While attractive, it is not part of the C family of languages and we had to drop it because Fortran90 adopts & for line continuation.

Because we are not compiler writers, we dropped a rarely used feature of Ratfor77 that was not easy for us to implement and is ugly anyway: Ratfor77 recognizes `break 2` which escapes from {{ }}.

Changing all the code that generated illustrations for four textbooks (of various ages) also turned up a few more issues: Fortran90 uses the words `scale` and `matmul` as intrinsics. Old Fortran77 programs using those words as variable names must be changed. Ratfor77 unwisely allowed variables of intrinsic (undeclared) types. We no

longer allow this. Ratfor90 forces `implicit none`.

New features in Ratfor90 are bracketed type, subroutine, function, and module procedures. In some ways this a further step towards the C, C++, Java model. It makes complicated modules, subroutines inside subroutines, and other advanced features of Fortran90 easier to interpret. Ratfor90 has better error messages than Ratfor77. Besides the use of stderr, a new file (`ratfor_problem`) marks the difficulty.

11.0.2 Examples

Below are simple Ratfor subroutines for erasing an array (`zero()`); (`null()`); for copying one array to another (`copy()`); for the signum function $sgn(x) = x/|x|$ (`signum()`); and (`tcaf`), a program using fortran90 modules and overloading to transient convolution.

11.0.3 Memory allocation in subroutines

For backward compatibility we allow the "temporary" memory allocation introduced by our Ratfor77 processor for example:

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

These declarations must follow other declarations and precede the executable statements. Automatic arrays are supported in Fortran90. To allow full code compatibility, `Ratfor90` simply translates this statement to

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

11.0.4 The main program environment

`Ratfor90` includes some traditional SEP local-memory-allocation and data-base-I/O statements that are described below. It calls an essential seplib initialization routine `initpar()`, organizes the self-doc, and simplifies data-cube input. The basic syntax for memory allocation is `allocate: real x(n1,n2)`. `Ratfor90` translates this syntax into a call to dynamically allocate a `allocatable` array. See the on-line self-documentation or the manual pages for full details. Following is a complete Ratfor 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
```

11.1 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.

Chapter 12

Entrance examination

1. (10 minutes) Given is 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 that m_1 and m_2 are already known. Derive a formula for finding m_3 that 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
```

Index

C, 1
C++, 1

examination for class entrance, 5

Fortran, 1

Java, 1

pitfall, 2

Ratfor, 1
Ratfor90, 1

SEP, 4