Short Note

RATional FORtran == Ratfor90

Robert G. Clapp and Jon F. Claerbout

keywords: computing

INTRODUCTION

Fortran is generally accepted as the most universal computer language for computational physics. However, for general programming, it has been surpassed by C. Ratfor is Fortran with C-like syntax. Ratfor was invented by the Kernighan and Plauger (1976), the same people who invented C. Ratfor uses C-like syntax, the syntax that is also found in the popular languages C++, Perl, and Java. Ratfor source is approximately 25-30% smaller than the equivalent Fortran source, so it is equivalently more readable.

Recently SEP has been shifting to the newest version of Fortran, Fortran90 (Clapp and Crawley, 1996; Fomel and Claerbout, 1996). Fortran90 allows for dynamic memory allocation and adds useful programming features such as structures, but still forces a verbose coding style. To take advantage of Fortran90’s new features, while maintaining the concise coding style provided by Ratfor, we wrote a new Ratfor preprocessor, Ratfor90, which produces Fortran90 rather than Fortran77 code. The newest Ratfor “compiler”, Ratfor90, is a simple word-processing program (written in Perl and freely distributed2) that inputs an attractive Fortran-like dialect and outputs Fortran90.

RATFOR BASICS

You should be able to read Ratfor if you already know Fortran, C, or any similar computer language. The Ratfor processor is not a compiler but a simple word-processing program that converts the Ratfor dialect to Fortran. To maximize your use of Ratfor, you will need to know its rules:

1email: bob@sep.stanford.edu jon@geo.stanford.edu
2http://sepwww.stanford.edu/src/ratfor90.html
<table>
<thead>
<tr>
<th>Function</th>
<th>Ratfor</th>
<th>Fortran90</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple statements on one line</td>
<td>May be separated by “;”.</td>
<td>Equivalent</td>
<td>Equivalent</td>
</tr>
<tr>
<td>do</td>
<td>Multi-line statements bracketed with { }.</td>
<td>DO/ END DO construct, may be named.</td>
<td>Equivalent</td>
</tr>
<tr>
<td>if</td>
<td>Multi-line statements bracketed { }</td>
<td>Multi-line require THEN/ END IF.</td>
<td>Equivalent</td>
</tr>
<tr>
<td>else/ else if</td>
<td>Multiple statements in { } single statements per construct do not require { }.</td>
<td>Requires THEN/ELSE THEN/ END IF construct</td>
<td>Equivalent</td>
</tr>
<tr>
<td>while</td>
<td>while() {}</td>
<td>DO WHILE()/END DO</td>
<td>Equivalent</td>
</tr>
<tr>
<td>break if/while</td>
<td>break</td>
<td>exit</td>
<td>Equivalent</td>
</tr>
<tr>
<td>iterate do</td>
<td>next</td>
<td>CYCLE</td>
<td>continue</td>
</tr>
<tr>
<td>relation operators</td>
<td>==, !=, &gt;, &lt;, &gt;=, &gt;</td>
<td>.eq. or ==, /= or .ne. .gt. or &gt;, &lt; or .lt.. .ge. or &gt;=, .le. or &lt;=</td>
<td>Equivalent</td>
</tr>
<tr>
<td>Comments</td>
<td>#, to the end of the is a comment</td>
<td>Same functionality with !.</td>
<td>enclosed by /* */</td>
</tr>
<tr>
<td>and and or</td>
<td>&amp;&amp; ,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>line continuation</td>
<td>-</td>
<td>&amp;</td>
<td>end of line delineated with “;”</td>
</tr>
<tr>
<td>for statement</td>
<td>for(initial; end; update)</td>
<td>Some of the functionality possible with DO.</td>
<td>Equivalent</td>
</tr>
</tbody>
</table>

**CHANGES FROM RATFOR77**

**Backward compatibility issues**

We were forced to make some changes to Ratfor77 because of new features 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: Ratfor77 recognizes break 2 which escapes from {{ } }. Breaking out of multiple loops is still possible using the loop naming feature provided by Fortran90.

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 implicit (undeclared) types. Ratfor90 includes an implicit none statement in all programs, eliminating a common
programming bug.

Extensions

New features in Ratfor90 are bracketed type, subroutine, function, where, and module statements. 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) indicates where problems with interpreting programs are encountered (see Appendix C).

In many geophysical applications we perform operations of the form:

\[ c(i1,i2) = c(i1,i2) + \text{scale} \times e(i1,i3,i4) \]

Because this type type of operation is so common we borrowed from C the += and -= operators, which changes the above line into:

\[ c(i1,i2) += \text{scale} \times e(i1,i3,i4) \]

SEP EXTENSIONS

The large amount of code written in Ratfor77 and SEP’s saw and sat pre-processors required that Ratfor90 handle the conventions that they introduced. By including the flag -sep on the command line Ratfor90 simulates the functions of saw and sat (memory allocation and parameter handling.)

Memory allocation

The main method at SEP for dynamic memory allocation under Ratfor77, saw, and sat was the allocate statement:

allocate: real x(n1,n2)

When Ratfor90 finds this statement, along with the corresponding main program/subroutine structure of saw and sat, it translates the allocate: statement into a Fortran allocatable array, allocates the array, and passes it, along with all other relevant variables to the subroutine.

In subroutines SEP allowed dynamic memory allocation through the use of the temporary keyword, for example:
temporal real*4 data(n1,n2,n3), convolution(j+k-1)

Automatic arrays are supported in Fortran90 so Ratfor90 simply translates this statement to:

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

Parameter handling

In addition saw and sat, and now Ratfor90, simplify parameter handling. Ratfor90 calls an essential SEPlib initialization routine initpar(), organizes the self-doc, and allows for various parameter handling keywords (from history, from par, from either, from aux, to aux, to history). For example, the line:

from either: integer n1,n2:ns,n3:nv=1

is translated into the much more verbose:

if (0==fetch('n1','d',n1)) then
   call sepperr('Could not obtain n1 from either')
end if
if (0/=putch('From either: n1','d',n1)) then
   call sepperr('trouble writing n1 to history file')
end if
if (0==fetch('n2','d',ns)) then
   call sepperr('Could not obtain n2 from either')
end if
if (0/=putch('From either: n2','d',ns)) then
   call sepperr('trouble writing n2 to history file')
end if
if (0==fetch('n3','d',nv)) then
   nv=1
end if
if (0/=putch('From either: n3','d',nv)) then
   call sepperr('trouble writing n3 to history file')
end if

REFERENCES


APPENDIX A

As an illustration, here is a simple program to convert from interval to RMS velocities in Ratfor90 and the corresponding Fortran90 code.

RATFOR90 CODE

```ratfor90
#!/usr/bin/env ratfor90
#dix
#
# Usage:
# dix <in.H> out.H
#
# Description
# Converts from interval to RMS velocity
#
# %end of self-documentation

program dix
    integer i1, i2, i3, n1, n2, n3
    real, allocatable, dimension(:, :) :: array
    real time, val, dt, dx

    from history: integer n1, n2, n3  # grab the size of the dataset
    # from the history file
    from history: real d1: dx  # get the sampling interval, store in dx
    allocate(array(n1, n2))
    do i3 = 1, n3
        call sread("in", array, n1*n2*4)
        array = 1./array  # Fortran90 array manipulation
        do i2 = 1, n2
            time = 0.; val = 0.
            do i1 = 1, n1
                dt = dx / array(i1, i2)
                val += dt * array(i1, i2)**2  # add sum Ratfor90 feature
                time += dt
                array(i1, i2) = sqrt(val/time)
            end do
        end do
        call srite("out", array, n1*n2*4)
    end do
} # bracketed programs
```

TRANSLATED FORTRAN90 CODE

! <
! dix
!
! Usage:
! dix <in.H >out.H
!
! Description
! Converts from interval to RMS velocity
!
% end of self-documentation
program dix
  implicit none
  integer i1,i2,i3,n1,n2,n3
  real, allocatable, dimension(:, :) :: array
  real time, val, dt, dx
  integer hetch, putch
  call initpar()
  call doc('/homes/sep/bob/papers/ratfor90/dix.rs90')
  if (0==hetch('n1','d',n1)) then
    call sepperr('Could not obtain n1 from history')
  end if
  if (0==putch('From history: n1','d',n1)) then
    call sepperr('trouble writing n1 to history file')
  end if
  if (0==hetch('n2','d',n2)) then
    call sepperr('Could not obtain n2 from history')
  end if
  if (0==putch('From history: n2','d',n2)) then
    call sepperr('trouble writing n2 to history file')
  end if
  if (0==hetch('n2','d',n2)) then
    call sepperr('Could not obtain n2 from history')
  end if
  if (0==putch('From history: n2','d',n2)) then
    call sepperr('trouble writing n2 to history file')
  end if
  ! from the history file
  if (0==hetch('','f',)) then
    call sepperr('Could not obtain from history')
  end if
  if (0==putch('From history: ','f',)) then
    call sepperr('trouble writing to history file')
  end if

end if
if (0==getch(’,’,’f’,)) then
   call sepperr(’Could not obtain  from history’) 
end if
if (0==putch(’From history: ’,’f’,)) then
   call sepperr(’trouble writing  to history file’) 
end if
allocate(array(n1,n2))
do i3=1,n3
   call sreed("in",array,n1*n2*4)
array=1./array  ! Fortran90 array manipulation
   do i2=1,n2
      time=0.
      val=0.
      do i1=1,n1
         dt=dx/array(i1,i2)
         val = val + dt*array(i1,i2)**2  ! add sum Ratfor90 feature
         time = time + dt
         array(i1,i2)=sqrt(val/time)
      end do
   end do
   call srite("out",array,n1*n2*4)
end do
end program
! bracketed programs
APPENDIX B

DOWNLOADING/INSTALLING

You can download Ratfor90 from http://sepwww.stanford.edu/sep/bob/ratfor90/ratfor90. You might need to change the first line of the code indicating where perl is on your system.

You can convert from Ratfor90 to Fortran90 on the command line by:

ratfor90 < input.r90 > output.f90

If you wish to use expanded SEP command line, history file manipulation, and self-documentation abilities add the `-sep -SOURCE /my/source/location` flags. An alternate approach is to use the SEP makefile rules which are explained in Reproducible electronic documents\(^3\). If you follow this approach you will need to:

- install ratfor90 in `/usr/local/bin/ratfor90` or edit the `Prg.defs.top` file.
- set the environmental variable `RATF90` to yes ( `setenv RATF90 yes`). If you are using the SEP setup you can add this line to your `Setup/cshrc.machinetype` [e.g. `cshrc.sgi, cshrc.i586, etc.`]
- name your Ratfor90 files:
  - using Fortran90 syntax `.r90`.
  - using Fortran90 and SEP allocation conventions `.rs90`.

By setting the environmental variable `RATF90`, SEP style ratfor77 code, normally indicated by the `.rt`, `.rs`, and `.rst` suffixes will converted to Fortran90 by Ratfor90 and compiled. These rules are in the updated version of the SEP makefile rules so if you have a previous version you will need to download a new one.

---

\(^3\)http://sepwww.stanford.edu/redoc
APPENDIX C

ERROR HANDLING

Ratfor90 creates a file called `ratfor_problem` whenever it encounters and error in the source code. The `ratfor_problem` file contains the processed source code, with the line that caused the processor problems clearly delineated. For example if you misspelled `if` in a source file:

```plaintext
    iff(a .eq. b){
```

and ran the Ratfor90 processor you would see:

**ERROR:**

Problem finding acceptable bracket statement
I was looking for `do, module, subroutine, etc` before a `
and couldn’t find it (spelling?????)
wrote file as far as I got to `ratfor_problem`

written to stderr and in the file `ratfor_problem`

**ERROR BEFORE ERROR**

```plaintext
    iff(a .eq. b){
```

**ERROR AFTER**

the location of the error is clearly indicated. The next time `ratfor90` is run, the `ratfor_problem` file is removed.