Programming Guidelines for NPARC Alliance Software Development*

Charles E. Towne
NASA Glenn Research Center
Cleveland, Ohio

1 Introduction

This document describes the programming guidelines to be used during NPARC Alliance software development projects.[1] It deals exclusively with Fortran 90, since most new NPARC Alliance software is written in that language.[2]

The guidelines are intended to enhance the following aspects of the final product, listed in decreasing order of importance:

- **Maintainability** — refers to how easy it is to understand the purpose of each element of the program, and to modify and extend the program.
- **Portability** — refers to how easily the program can be ported to new computational platforms.
- **Efficiency** — refers to the amount of computer resources (CPU time, memory, disk storage, etc.) required to run the program.

Much of this material has been taken, sometimes verbatim, from the following documents available on the World-Wide Web:

- Levine, David L., "Fortran 77 Coding Guidelines," https://groups.google.com/forum/#!searchin/comp.lang.fortran/fortran$2077$20style$20guide|sort:relevance/comp.lang.fortran/6sZH408DM4o/ISkKZzqGGxIJ
- Huddleston, John, "Fortran Coding Standard"
- Robey, Tom, and Smith, Brian, eds., "Future of Fortran - Moving from F77"
- Andrews, Phillip; Cats, Gerard; Dent, David; Gertz, Michael; and Ricard, Jean Louis, "European Standards For Writing and Documenting Exchangeable Fortran 90 Code," http://research.metoffice.gov.uk/research/nwp/numerical/fortran90/f90_standards.html

*This is version 2.0 of this document, released 22 Apr 2004.

[1]Since much of the NPARC Alliance software has been developed over several years, some of the programming guidelines described in this document will not necessarily be followed throughout the final product. They should, however, be followed for any new code that is written.

[2]Throughout this document, the term "Fortran" should be understood to mean Fortran 90.

2 Program Development and Design

Items in this section are fairly general and fundamental in nature. They impact all three of the items listed above — maintainability, portability, and efficiency.

2.1 Language

Use ANSI standard Fortran 90 exclusively, with the following exceptions:

- ANSI C code may be used where Fortran 90 is inadequate.
- Existing low-level library routines written in Fortran 77 may be used.

2.2 Organization

- Write modular code.
- In general, put each subprogram in a separate file, using the subprogram name as the file name, with a *.f90* extension.
- Within each routine, use `interface` blocks to explicitly specify the interface to called routines. In Wind-US, `interface` blocks for all routines are stored in modules in the *interfaces* directory.
- Group related files in a single directory.
- Names of files and directories should reflect their purpose.

2.3 Common Blocks

- Don't use blank common.
- Put all common blocks in include files, one per file, using the common block name as the file name, with a `.inc` extension.
- Strike a common-sense balance on the number of common blocks. Group related variables together in separate common blocks, but minimize the total number of blocks.
- `Save` all common blocks, in the include file.

2.4 Data Types

- Use `Implicit none` in each program unit, and explicitly declare all variables and parameters. Common variables and parameters should be declared in the relevant include file.
- In general, use `kind` parameters to declare variable types, especially for floating point data. For Wind-US, the necessary `kind` parameters are defined in module `data_types`.
- Don't use `*`'ed forms, like `Real*8`.
- Don't compare arithmetic expressions of different types; convert the type explicitly.

2.5 Dynamic Memory

- Assign memory for arrays dynamically, using automatic arrays, allocatable arrays, and/or array pointers. Explicitly deallocate memory used by allocatable arrays and array pointers when they're no longer needed. In Wind-US, use the generic routines `alloc` and `dealloc` for managing dynamic memory.

2.6 Compilation

- Use a makefile for routine compiling and linking.
- Avoid embedding compiler directives in the code, unless an equivalent compiler option is not available. In that case, include comments describing its effect, the target machine and operating system, and the compiler.
- During development, low-level-optimization, no-optimization, and/or special debugging compiler options are often used. Before releasing a code modification, re-test it with the same level of optimization that will be used for the final production code.
- Before releasing a code modification, re-test it using compiler options that flag various run-time error conditions. Examples include options to detect attempts to use variables that haven't been assigned a value, or arrays with out-of-range indices.

3 Coding Style

Items in this section are fairly specific, and primarily impact the readability, and thus the maintainability, of the final product. It is recognized that rules for "good coding style" are somewhat subjective.[3] Some flexibility for personal preference should be acceptable. However, these guidelines should be followed at least in spirit throughout the program.

3.1 Program Units

- Begin main programs with a `Program` statement.
- Don't use multiple entries or alternate returns.
- Use the `intent` attribute in the type declaration statement for dummy variables.
- Match the arguments in the calling (sub)program to those of the called subprogram in both number and type.
- Use the following order for statements within each subprogram:
  - Standard header section
  - Use modules
  - Parameter definitions
  - Common blocks
  - Type declarations for subprogram arguments

[3] The guidelines in this document no doubt reflect some of my own biases.

  - Type declarations for local variables
  - Executable code
- Functions should not have side effects. (I.e., don't change the arguments or any common variables inside the function.)
- Use generic names for library functions, rather than precision-specific ones.
- Name external functions in an `External` statement.

3.2 Statement Form

- Use free-form formatting, but for readability:
  - Keep line lengths below 80 characters.
  - Start each line in column 7 or higher.
  - Reserve columns 1–5 for statement labels.
  - Don't use the optional continuation character (i.e., `&`) at the start of continuing lines. Avoid splitting keywords and character strings between lines.

  Note that with free-form formatting, an `&` must be the last character (except for comments) in a line that is to be continued.
- Split long lines before or after an operator, preferably a `+` or `-`.
- Don't write more than one statement per line.

3.3 Statement Labels

- Minimize the use of statement labels, where appropriate.
- Don't use unreferenced labels.
- Use labels in ascending order.

3.4 Upper/Lower Case

- Use upper case for parameters, lower case with an initial capital letter for Fortran keywords, and lower case for everything else except comments and character strings.
- Write comments as normal text, with normal capitalization rules.

3.5 Spacing

- Use spacing to enhance readability.
- Indent contents of code blocks (i.e., `do` loops, block `if`, etc.). Suggested amount is three spaces.
- Don't use tabs.

- Use spacing in equations to clarify precedence of operators. I.e., normally put one space on either side of `=`, `+`, and `-` operators (except in subscripts), but none around `*`, `/`, or `**` operators. For example, this:

  ```
  y1 = (-b + Sqrt(b**2 - 4.*a*c))/(2.*a)
  ```

  is easier to read than this:

  ```
  y1=(-b+Sqrt(b**2-4.*a*c))/(2.*a)
  ```

  or this:

  ```
  y1 = ( - b + Sqrt ( b ** 2 - 4. * a * c ) ) / ( 2. * a )
  ```

- Use spacing to reveal patterns in continuation lines and in separate but logically related statements. For example, this:

  ```
  dum1 = Sqrt((fr (i,j) - fr ( 1, j))**2 + &
              (fth(i,j) - fth( 1, j))**2)
  dum2 = Sqrt((fr (i,j) - fr (n1, j))**2 + &
              (fth(i,j) - fth(n1, j))**2)
  dum3 = Sqrt((fr (i,j) - fr ( i, 1))**2 + &
              (fth(i,j) - fth( i, 1))**2)
  dum4 = Sqrt((fr (i,j) - fr ( i,n2))**2 + &
              (fth(i,j) - fth( i,n2))**2)
  ```

  is easier to read than this:

  ```
  dum1 = Sqrt((fr(i,j) - fr(1,j))**2 + (fth(i,j) - &
  fth(1,j))**2)
  dum2 = Sqrt((fr(i,j) - fr(n1,j))**2 + (fth(i,j) - &
  fth(n1,j))**2)
  dum3 = Sqrt((fr(i,j) - fr(i,1))**2 + (fth(i,j) - &
  fth(i,1))**2)
  dum4 = Sqrt((fr(i,j) - fr(i,n2))**2 + (fth(i,j) - &
  fth(i,n2))**2)
  ```

3.6 Variable Names

- Use names that are descriptive of the entity being represented, and/or are consistent with the standard notation in the field.
- In general, follow standard Fortran convention for the variable type. I.e., integers start with `i`, `j`, `k`, `l`, `m`, or `n`, all others are real.
- Don't use keyword, subprogram, or common block names for variables.
- Don't give a local variable the same name as any common variable.

3.7 Arrays

- Dimension arrays in the type declaration statement, not in a common block or a separate `Dimension` statement.
- When passing character variables into a subprogram, use the assumed-length form in the type declaration statement inside the subprogram. I.e.,

```
Subroutine sub (c)
Character*(*) c
```

- Don't exceed the bounds of the array dimensions.
- Begin array indices with 1, unless there's a *very* good (and well-commented) reason to do otherwise.

3.8 Common Blocks

- Don't include common blocks in subprograms where they aren't needed.

3.9 Fortran Parameters

- Use `Parameter` statements to symbolically name constants that may change from compilation to compilation, or that are long and susceptible to typing errors.
- Use `Parameter` statements to symbolically name array dimensions that may change from compilation to compilation.
- `Parameter` statements should be in separate include files or modules.

3.10 Control Statements

- Short do loops may be written using simple `Do` and `End do` statements, without labels.
- Long do loops and if blocks (more than a page or so), should mark the end of the construct in some way that "connects" it with the start. One convenient and readable method is to use an in-line comment on the ending statement that repeats the beginning statement. E.g.,

  ```
  If ( bccode == 13 ) then

     [Lines and lines of code]

     Do i = 1,nzones
        If ( zondim(1,i) > 0 ) then

           [<i>More lines and lines of code</i>]

        End if   ! If ( zondim(1,i) > 0 ) then
     End do      ! Do i = 1,nzones
  End if         ! If ( bccode == 13 ) then
  ```

  For do loops, another method is using a Fortran 77 style loop, ending with a labeled `Continue` statement.
- Minimize the use of `Goto` statements, especially where they can be replaced by short'ish if blocks, but don't create convoluted code just to avoid using them. Don't be afraid to use a `Goto` where it makes sense. An example might be a long (more than a page) conditional section of code. In this case a well-commented `Goto` block, which ends with an easily-noticed statement label, may be more readable than an indented if block without an ending statement label. Also consider making a long conditional section a separate subprogram.

3.11 Comments

- Use comments liberally to describe what's being done. Where code may be confusing, use longer comments to describe why something's being done the way that it is.
- Make each comment meaningful; don't simply re-iterate what's already obvious from the coding itself. As an obvious example, this:

  ```
  !-----Get the turbulent viscosity using Baldwin-Lomax model
        Call turbbw
  ```

  is more meaningful than this:

  ```
  !-----Call turbbw
        Call turbbw
  ```

- Use a consistent method to help the reader distinguish comments from code, such as the "`---`" leaders in the examples above.
- Start the text of comments at the same indentation level as the code being described.
- Use a standard header section at the beginning of each subprogram defining its purpose.
- Use in-line comments, with `!` as the delimiter, where appropriate for short explanations or clarifications. Start in-line comments far enough to the right (e.g., three spaces or more from the end of the statement) to help distinguish comments from code. Where appropriate, align them vertically with nearby in-line comments.
- Define each common block variable using an in-line comment on its type statement in the include file. Each common variable will thus have a separate type statement.
- Define key local variables using in-line comments on the type statements in the subprogram.

3.12 Input/Output

- Open output files with `Status='unknown'` where possible.
- Use the input error recovery parameter `iostat`.
- Place once-used `Format` statements immediately following their reference. Place those used more than once at the end of the subprogram.

3.13 Obsolete/Forbidden Features

The following Fortran features are either formally declared as obsolete, or widely considered to be poor programming practice, and should not be used:

- Arithmetic if statements
- Do loops with non-integer indices
- Shared do loop termination statements
- `Pause` statements
- Assigned and computed `Go to` statements
- Hollerith edit descriptors and Hollerith character strings

- `Equivalence` statements
- Alternate return statements

Appendix — Standard Subprogram Format

The following is an example illustrating the use of the guidelines in this document for one of the subprograms in Wind-US. First, here's an include file named *mxdim.par* that's needed:

```
Integer MXDIM   ! Max grid points in any direction
Parameter (MXDIM = 1023)
```

And here's another named *test.inc*:

```
Common /test/ itest
Integer itest(200)   ! Test options
Save test
```

And here's the subprogram itself, named *blomax.f90*:

```
!
!-----Purpose:  This subroutine computes the turbulent viscosity
!               coefficient using the Baldwin-Lomax model.
!

!-----Use modules
      Use data_types   ! kind parameters

!-----Require explicit typing of variables
      Implicit none

!-----Parameter statements
      Include 'mxdim.par'

!-----Common blocks
      Include 'test.inc'

!-----Input arguments
      Integer(kindInt), intent(in) :: jedge       ! Index of bl edge

      Real(kindSingle), intent(in) :: re          ! Reference Reynolds number
      Real(kindSingle), intent(in) :: yy(MXDIM)   ! Distance from wall
      Real(kindSingle), intent(in) :: rh(MXDIM)   ! Static density
      Real(kindSingle), intent(in) :: uu(jedge)   ! Velocity
      Real(kindSingle), intent(in) :: vo(MXDIM)   ! Vorticity
      Real(kindSingle), intent(in) :: vi(MXDIM)   ! Laminar viscosity coeff
      Real(kindSingle), intent(in) :: tauw        ! Shear stress at the wall

!-----Output arguments
      Real(kindSingle), intent(out) :: tv(jedge)  ! Turbulent viscosity coeff

!-----Local variables
      Integer(kindInt) icross   ! Flag for inner/outer region
```

```
      Integer(kindInt) j        ! Do loop index
      Integer(kindInt) jedg     ! Index limit for Fmax search

      Real(kindSingle) al       ! Mixing length
      Real(kindSingle) aplus    ! Van Driest damping constant
      Real(kindSingle) bigk     ! Clauser constant
      Real(kindSingle) ccp      ! Constant in outer region model
      Real(kindSingle) ckleb    ! Constant in Klebanoff intermittency factor
      Real(kindSingle) cwk      ! Constant in outer region model
      Real(kindSingle) fkleb    ! Klebanoff intermittency factor
      Real(kindSingle) fl       ! Baldwin-Lomax F parameter
      Real(kindSingle) fmax     ! Baldwin-Lomax Fmax parameter
      Real(kindSingle) frac     ! Fractional decrease in F req'd for peak
      Real(kindSingle) fwake    ! Baldwin-Lomax Fwake parameter
      Real(kindSingle) rdum     ! Ratio of distance from wall to ymax
      Real(kindSingle) smlk     ! Von Karman constant
      Real(kindSingle) tvi      ! Inner region turbulent viscosity coeff
      Real(kindSingle) tvo      ! Outer region turbulent viscosity coeff
      Real(kindSingle) udif     ! Max velocity difference
      Real(kindSingle) umax     ! Max velocity
      Real(kindSingle) umin     ! Min velocity
      Real(kindSingle) ymax     ! Distance from wall to location of Fmax
      Real(kindSingle) yp       ! y+
      Real(kindSingle) ypcon    ! Coeff term for y+, based on wall values
      Real(kindSingle) ypconl   ! Coeff term for y+
      Real(kindSingle) yyj      ! Distance from wall

!-----Set constants

      aplus = 26.
      ccp   = 1.6
      ckleb = 0.3
      cwk   = 0.25
      smlk  = 0.4
      bigk  = 0.0168
      If (itest(126) == 1) bigk = 0.0180   ! Comp. correction (cfl3de)

!-----Compute stuff needed in model

!-----Get coefficient term for y+
      If (itest(25) == 1) then     ! Using wall vorticity as in cfl3de
         ypcon = Sqrt(re*rh(1)*vo(1)/vi(1))
      Else                         ! Using wall shear stress
         If (tauw <= 1.e-9) tauw = 1.e-9
         ypcon = Sqrt(re*rh(1)*tauw)/vi(1)
      End if

!-----Set index limit for Fmax search, and fractional decrease needed to
!-----   qualify as first peak
      jedg = jedge
      frac = .70
```

```
      If (itest(163) > 0) then           ! User-spec. frac. decrease
         frac = Real(itest(163))/1000.
      Else if (itest(163) < 0) then      ! Reset search range, use max
         jedg = Min(jedge,-itest(163))   !    value, not first peak
         frac = 0.0
      Endif

!-----Get max velocity and max velocity difference
      umin = 0.
      umax = 0.
      Do j = 2,jedge
         umax = Max(umax,uu(j))
         umin = Min(umin,uu(j))
      End do
      udif = umax - umin

!-----Get Fmax by searching for first peak in F

      ymax = 0.
      fmax = 0.
      ypconl = ypcon
      Do j = 2,jedg
         yyj = yy(j)
         If (itest(26) == 1) then   ! Use local values in y+
            ypconl = ypcon*Sqrt(rh(j)/rh(1))*vi(1)/vi(j)
         End if
         yp = ypconl*yyj   ! y+
         fl = yyj*vo(j)*(1. - Exp(-yp/aplus))   ! B-L F parameter
         If (fl > fmax) then             ! Set new Fmax
            fmax = fl
            ymax = yyj
         Else if (fl > frac*fmax) then   ! Keep searching
            Cycle
         Else                            ! Found Fmax, so get out
            Exit
         End if
      End do

!-----Reset ymax and Fmax if necessary, to avoid overflows
      If (ymax < 1.e-6) ymax = 5.e-5
      If (fmax < 1.e-6) fmax = 5.e-5

!-----Compute turbulent viscosity

      icross = 0
      ypconl = ypcon
      Do j = 2,jedge
         yyj = yy(j)
         tvi = 1.e10

!--------Inner region value, if we're still there
```

```
         If (icross == 0) then
            If (itest(26) == 1) then   ! Use local values in y+
               ypconl = ypcon*Sqrt(rh(j)/rh(1))*vi(1)/vi(j)
            End if
            yp = ypconl*yyj   ! y+
            al = smlk*yyj*(1. - Exp(-yp/aplus))   ! Mixing length
            tvi = rh(j)*al*al*vo(j)
         End if

!--------Outer region value
         rdum = yyj/ymax
         If (rdum >= 1.e5) then     ! Prevent overflow
           fkleb = 0.0
         Else                       ! Klebanoff intermittency factor
           fkleb = 1./(1. + 5.5*(ckleb*rdum)**6)
         End if
         fwake = Min(ymax*fmax,cwk*ymax*udif*udif/fmax)
         tvo = bigk*ccp*rh(j)*fwake*fkleb

!--------Set turbulent viscosity, plus flag if we're in outer region
         tv(j) = tvi
         If (tvo < tvi) then
            icross = 1
            tv(j) = tvo
         End if

!--------Non-dimensionalize
         tv(j) = re*tv(j)
      End do   ! Do j = 2,jedge

!-----Zero out turbulent viscosity at wall
      tv(1) = 0.0

      Return
      End
```