ParaFEM Coding Standard for Fortran 90

This standard has been prepared by Lee Margetts at the University of Manchester. It is based on Version 1.1 of the *European Standards for Writing and Documenting Exchangeable Fortran 90 Code* published by the Met Office.

Contents

1.0 Introduction

The aim of this document is to provide a framework for the use of Fortran 90 in ParaFEM and related projects and thereby to facilitate the exchange of code between them. In order to achieve this goal we set standards for the documentation of code, both internal and external to the software itself, as well as setting standards for writing the code. The latter standards are designed to improve code readability and maintainability as well as to ensure, as far as possible, its portability and the efficient use of computer resources.

Back to top

2.0 Documentation

Documentation may be split into two categories: external documentation, outside the code; and internal documentation, inside the code. These are described in sections 2.1 and 2.2 respectively. In order for the documentation to be useful it needs to be both up to date and readable at centres other than that at which the code was originally produced. Since these other centres may wish or need to modify the imported code we specify that all documentation, both internal and external, must be available in English.

2.1 External Documentation

In most cases this will be provided at the package level, rather than for each individual routine. It must include the following:

- Top Level Scientific documentation: this sets out the problem being solved by the package and the scientific rationale for the solution method adopted. This documentation should be independent of (i.e. not refer to) the code itself.

- Implementation documentation: this documents a particular implementation of the solution method described in the scientific documentation. All routines (subroutines, functions, modules etc...) in the package should be listed by name together with a brief description of what they do.

- A User Guide: this describes in detail all inputs into the package. This includes both subroutine arguments to the package and any switches or 'tuneable' variables within the package. Where appropriate default values; sensible value ranges; etc should be given. Any files or namelists read should be described in detail.

2.2 Internal Documentation

This is to be applied at the individual routine level. There are four types of internal documentation, all of which must be present.

- Procedure headers: every subroutine, function, module etc must have a header. The purpose of the header is to describe the function of the routine, probably by referring to external documentation, and to document the variables used within the routine. All variables used within a routine must be declared in the header and commented as to their purpose. It is a requirement of this standard that the headers listed in Appendix A be used and completed fully. Developers are allowed to add extra sections to these headers if they so wish.

- Section comments: these divide the code into numbered logical sections and may refer to the external documentation. These comments must be placed on their own lines at the start of the section they are defining. The recommended format for section comments is:

  ```
  !---------------------------------------------
  ! [Section number] [Section title]
  !---------------------------------------------
  ```

  where the text in [] is to be replaced appropriately.

- Other comments: these are aimed at a programmer reading the code and are intended to simplify the task of understanding what is going on. These comments must be placed on the line either immediately before or after the code they are commenting. The recommended format for these comments is:

  ```
  ! [Comment]
  ```

  where the text in [] is to be replaced appropriately.

- Meaningful names: code is much more readable if meaningful words are used to construct variable & subprogram names.
- It is a requirement that all internal documentation be written in English.

  Back to top

3.0 Coding Rules For Packages

These rules are loosely based on the "plug compatibility" rules of Kalnay et al. (1989). Rules which appear elsewhere in this document have not been duplicated in this section.

- A package shall refer only to its own modules and subprograms and to those intrinsic routines included in the Fortran 90 standard.

- A package shall provide separate set up and running procedures, each with a single entry point. All initialization of static data must be done in the set up procedure and this data must not be altered by the running procedure.

- External communication with the package must be via:

  - The argument lists of the packages entry and set up routines.

  - A NAMELIST file read.

  - Environment variables read by a set of standard routines. The environment variables must be documented in and set by a script appropriate to the operating system in use (i.e. a posix script for the unix operating system).

- Internally the package may use Modules - for example the set up and running procedures may communicate in this way.

- Interface blocks must be provided for the set up and running procedures (possibly via module (s)). This allows the use of assumed shape arrays, optional arguments, etc as well as allowing the compiler to check that the actual and dummy arguments types match. If variables in these external argument lists are of derived type, a module must be supplied which contains the type definition statements required to make these derived types available to the routine calling the package.

- The package shall not terminate program execution. If any error occurs within the package it should gracefully exit, externally reporting the error via an integer variable in its argument list (+ve = fatal error). Packages should also write a diagnostic message describing the problem, using fortran I/O, to an 'error stream' unit number selectable via the package's set up routine.

  Note that if the package starts at the unix script, rather than Fortran, level making a graceful exit includes returning control to the package's script level by using a STOP statement in the Fortran part of the package.

- Precompilers: these are used, for example, to provide a means of selecting (or deselecting) parts of the code for compilation. Clearly to simplify portability of code we need to all use the same precompiler, and this needs to be available to every centre. The C precompiler is probably the best option since it will be found on all machines using the unix operating system.

- All unix scripts must be written using the posix shell. This is a standardized shell, available on all POSIX compliant unix systems, with many useful features.

- Each program unit should be stored in a separate file.

Back to top

4.0 Guidance For The Use Of Dynamic Memory

The use of dynamic memory is highly desirable as, in principle, it allows one set of compiled code to work for any specified model size (or at least up to hardware memory limits); and allows the efficient reuse of work space memory. Care must be taken, however, as there is potential for inefficient memory usage, particularly in parallelized code. For example heap fragmentation can occur if space is allocated by a lower level routine and then not freed before control is passed back up the calling tree. There are three ways of obtaining dynamic memory in Fortran 90:

1. Automatic arrays: These are arrays initially declared within a subprogram whose extents depend upon variables known at runtime e.g. variables passed into the subprogram via its argument list.

2. Pointer arrays: Array variables declared with the POINTER attribute may be allocated space at run time by using the ALLOCATE command.

3. Allocatable arrays: Array variables declared with the ALLOCATABLE attribute may be allocated space at run time by using the ALLOCATE command. However, unlike pointers, allocatables are not allowed inside derived data types.

4. For small arrays, automatic arrays may be used in preference to the other forms of dynamic memory allocation.

5. Space allocated using 2) and 3) above must be explicitly freed using the DEALLOCATE statement.

6. In a given program unit do not repeatedly ALLOCATE space, DEALLOCATE it and then ALLOCATE a larger block of space. This will almost certainly generate large amounts of unusable

memory.

7. Always test the success of a dynamic memory allocation and deallocation. The ALLOCATE and DEALLOCATE statements have an optional argument to let you do this.

Back to top

5.0 Coding Rules For Routines

By routines we mean any fortran program unit such as a subroutine, function, module or program. These rules are designed to encourage good structured programming practice, to simplify maintenance tasks, and to ease readability of exchanged code by establishing some basic common style rules.

Back to top

5.1 Banned Fortran Features

Some of the following sections detail features deprecated in or made redundant by Fortran 90. Others ban features whose use is deemed to be bad programming practice as they can degrade the maintainability of code.

- COMMON blocks - use Modules instead.
- EQUIVALENCE - use POINTERS or derived data types instead.
- Assigned and computed GO TOs - use the CASE construct instead.
- Arithmetic IF statements - use the block IF construct instead.
- Labels (only one allowed use).
  - Labelled DO constructs - use End DO instead.
  - I/O routine's End = and ERR = use IOSTAT instead.
  - FORMAT statements: use Character parameters or explicit format specifiers inside the Read or Write statement instead.
  - GO TO

    The only recognized use of GO TO, indeed of labels, is to jump to the error handling section at the end of a routine on detection of an error. The jump must be to a CONTINUE statement and the label used must be 9999. Evens so, it is recommended that this practice be avoided.

    Any other use of GO TO can probably be avoided by making use of IF, CASE, DO WHILE, EXIT or CYCLE statements. If a GO TO really has to be used, then clearly comment it to explain what is going on and terminate the jump on a similarly commented CONTINUE statement.
- PAUSE
- ENTRY statements: - a subprogram may only have one entry point.
- Functions with side effects i.e. functions that alter variables in their argument list or in modules used by the function; or one that performs I/O operations.
  This is very common in C programming, but can be confusing. Also, efficiencies can be made if the compiler knows that functions have no side effects. High Performance Fortran, a variant of Fortran 90 designed for massively parallel computers, will allow such instructions.

- Implicitly changing the shape of an array when passing it into a subroutine. Although actually forbidden in the standard it was very common practice in FORTRAN 77 to pass 'n' dimensional arrays into a subroutine where they would, say, be treated as a 1 dimensional array. This practice, though banned in Fortran 90, is still possible with external routines for which no Interface block has been supplied. This only works because of assumptions made about how the data is stored: it is therefore unlikely to work on a massively parallel computer. Hence the practice is banned.

Back to top

5.2 Style Rules

The general ethos is to write portable code that is easily readable and maintainable. Code should be written in as general a way as possible to allow for unforseen modifications. In practice this means that coding will take a little longer. This extra effort is well spent, however, as maintenance costs will be reduced over the lifetime of the software.

- Use free format syntax.
- The maximum line length permitted is 80 characters. Fortran 90 allows a line length of up to 132 characters, however this could cause problems when viewed on older terminals, or if print outs have to be obtained on A4 paper.

- Implicit none must be used in all program units. This ensures that all variables must be explicitly declared, and hence documented. It also allows the compiler to detect typographical errors in variable names.

- The first statements after the program name and header should be idented two characters.

- Use meaningful variable names, preferably in English. Recognized abbreviations are acceptable as a means of preventing variable names getting too long.

- Fortran statements must be written in upper case only. Names, of variables, parameters, subroutines etc may be written in mixed, mostly lower, case.

- ParaFEM function names that are "parallel versions" of FORTRAN intrinsics must be written in upper case only (e.g. SUM_P)

- To improve readability indent code within DO; DO WHILE; block IF; CASE; Interface; etc constructs by 2 characters.

- Indent continuation lines to ensure that e.g. parts of a multi line equation line up in a readable manner.

- Where they occur on separate lines indent internal comments to reflect the structure of the code.

- Use blank space, in the horizontal and vertical, to improve readability. In particular leave blank space between variables and operators, and try to line up related code into columns. For example,

  Instead of:

```
! Initialize Variables

       x=1

       MEANINGFUL_NAME=3.0

       SILLY_NAME=2.0
```

  Write:

```
       ! Initialize Variables
```

```
x = 1

MeaningfulName = 3.0

SillyName      = 2.0
```

Similarly, try to make equations recognizable and readable as equations. Readability is greatly enhanced by starting a continuation line with an operator placed in an appropriate column rather than ending the continued line with an operator.

- Do not use tab characters in your code: this will ensure that the code looks as intended when ported.
- Separate the information to be output from the formatting information on how to output it on I/O statements. That is don't put text inside the brackets of the I/O statement.
- Delete unused header components.

Back to top

5.3 Use Of New Fortran Features

It is inevitable that there will be 'silly' ways to use the new features introduced into Fortran 90. Clearly we may want to ban such uses and to recommend certain practices over others. However, there will have to be a certain amount of experimentation with the new language until we gain enough experience to make a complete list of such recommendations. The following rules will have to be amended and expanded in the light of such experience.

- We recommend the use of Use, ONLY to specify which of the variables, type definitions etc defined in a module are to be made available to the Useing routine.
- Discussion of the use of Interface Blocks:

  Introduction

  Explicit interface blocks are required between f90 routines if optional or keyword arguments are to be used. They also allow the compiler to check that the type, shape and number of arguments specified in the CALL are the same as those specified in the subprogram itself. In addition some compilers (e.g. the Cray f90 compiler) use the presence of an interface block in order to determine if the subprogram being called is written in f90 (this alters how array information is passed to the subroutine). Thus, in general it is desirable to provide explicit interface blocks between f90 routines.There are several ways to do this, each of which has implications for program design; code management; and even configuration control. The three main options are discussed in the following sections:

  Option I: Explicitly Coded Interface Blocks

  Interface blocks may be explicitly written into the calling routine, essentially by copying the argument list declaration section from the called routine. This direct approach has, however, some disadvantages. Firstly, it creates an undesirable increase in the work required to maintain the calling routine, as if the argument list of the called routine changes the Interface block must be updated as well as the CALL. Further, there is no guarantee that the Interface block in the calling routine is actually up to date and the same as the actual interface to the called routine.

  Option II: Explicitly Coded Interface Blocks in a Module

  Interface blocks for all routines in a package may be explicitly written into a module, and this module used by all routines in the package. This has the advantage of having one routine to examine to find the interface specification for all routines - which may be easier than individually examining the source code for all called routines. However, an Interface block must still be maintained in addition to the routine itself and CALLs to it, though a program or e.g. a unix script could be written to automatically generate the module containing the interface blocks.

Option III: Automatic Interface Blocks

Fortran 90 compilers can automatically provide explicit interface blocks between routines following a Contains statement. The interface blocks are also supplied to any routine Useing the module. Thus, it is possible to design a system where no Interface blocks are actually coded and yet explicit interface blocks are provided between all routines by the compiler. One way to do this would be to 'modularise' the code at the f90 module level, i.e. to place related code together in one module after the Contains statement. Routine a, in module A calling routine b in module B would then only have to Use module B to be automatically provided with an explicit interface to routine b. Obviously if routine b was in module a instead then no Use would be required. One consequence of this approach is that a module and all the routines contained within it make up a single compilation unit. This may be a disadvantage if modules are large or if each module in a package contains routines which Use many other modules within the package (in which case changing one routine in one module would necessitate the recompilation of virtually the entire package). On the other hand the number of compilation units is greatly reduced, simplifying the compilation and configuration control systems.

Conclusion

Options II) and III) both provide workable solutions to the problem of explicit interface blocks. Option III is probably preferable as the compiler does the work of providing the interface blocks, reducing programming overheads, and at the same time guaranteeing that the interface blocks used are correct. Which ever option is chosen will have significant impact on code management and configuration control as well as program design.

- Subroutine arguments should be ordered in a logical and consistent manner. The suggested scheme is to place those arguments with INTENT(IN) first, then INTENT(INOUT) followed by INTENT (OUT). It is convenient to order the arguments alphabetically for each INTENT, e.g.:

```
SUBROUTINE READALL_ELS_BINARY_FNAME(fname,iel_start,ndim,nels,nn, &
                                    npes,numpe,g_num_pp)

rather than

SUBROUTINE READALL_ELS_BINARY_FNAME(fname,g_num_pp,nn,ndim,nels, &
                                    iel_start,numpe,npes)
```

- Array notation should be used whenever possible. This should help optimization and will reduce the number of lines of code required. To improve readability show the array's shape in brackets, e.g.:

```
1dArrayA(:)    = 1dArrayB(:) + 1dArrayC(:)

2dArray(:, :) = scalar * Another2dArray(:, :)
```

- When accessing subsections of arrays, for example in finite difference equations, do so by using the triplet notation on the full array, e.g.:

```
2dArray(:, 2:len2) = scalar                         &

                   * ( Another2dArray(:, 1:len2 -1) &

                     - Another2dArray(:, 2:len2)    &
```

- Always name 'program units' and always use the End program; End subroutine; End interface; End module; etc constructs, again specifying the name of the 'program unit'.

- Use >, >=, ==, <, <=, /= instead of .gt., .ge., .eq., .lt., .le., .ne. in logical comparisons. The new syntax, being closer to standard mathematical notation, should be clearer.

- Don't put multiple statements on one line: this will reduce code readability.

- Variable declarations: it would improve understandability if we all adopt the same conventions for declaring variables as Fortran 90 offers many different syntaxes to achieve the same result.

  - Don't use the DIMENSION statement or attribute: declare the shape and size of arrays inside brackets after the variable name on the declaration statement.
  - Always use the :: notation, even if their are no attributes.
  - Declare the length of a character variable using the (len =) syntax.

- We recommend against the use of recursive routines on efficiency grounds (they tend to be inefficient in their use of cpu and memory).
- It is recommended that new operators be defined rather than overload existing ones. This will more clearly document what is going on and should avoid degrading code readability or maintainability.
- To improve portability between 32 and 64 bit platforms, it is extremely useful to make use of kinds to obtain the required numerical precision and range. A module should be written to define parameters corresponding to each required kind, for example:

```
Integer, parameter       :: single            & ! single precision kind.
                          = selected_real_kind(6,50)

Integer, parameter       :: double            & ! double precision kind.
                          = selected_real_kind(12,150)
```

This module can then be Used in every routine allowing all variables declared with an appropriate kind e.g.

```
Real(single), pointer   :: geopotential(:,:,:) ! Geopotential height
```

Back to top

6.0 Enforcing These Standards

It is obviously important to ensure that these standards are adhered to - particularly that the documentation is kept up to date with the software; and that the software is written in as portable a manner as possible. If these standards are not adhered to exchangeability of the code will suffer.

Back to top

References

Kalnay et al. (1989) "Rules for Interchange of Physical Parametrizations" Bull. A.M.S., 70 No. 6, p 620.

Back to top

Appendix A: Modification History:

31/12/07: Version 1.0 Lee Margetts (University of Manchester)
16/03/10: Version 1.1 Lee Margetts (University of Manchester)

Appendix B: Standard Headers

The standard headers to be used are those provided by ROBODOC. The format of the headers enables the automatic generation of documentation in HTML, LaTex or Postscript format.

Back to top

Appendix C: Examples

```
SUBROUTINE READALL_ELS_BINARY_FNAME(fname,iel_start,ndim,nels,nn, &
                                    npes,numpe,g_num_pp)

 !/****f* devel_mpi/readall_els_binary_fname
 !*  NAME
 !*    SUBROUTINE: readall_els_binary_fname
 !*  SYNOPSIS
 !*    Usage:      CALL readall_els_binary_fname(fname,iel_start,ndim,nels,nn, &
 !*                                              npes,numpe,g_num_pp
 !*  FUNCTION
 !*    Read the node numbers for each element from a binary file and
 !*    distribute the data to the appropriate processor.
 !*
 !*    The strategy is to send all the data to all the processors.
 !*    Afterwards each processor records only what it is interested in.
 !*
 !*    1. The node numbers for each element in the mesh are read from the
 !*       file FNAME into a global array G_NUM(NOD,NELS) by the processor
 !*       of rank NPES-1. G_NUM has the dimensions NOD (number of nodes per
 !*       element) and NELS (total number of elements in the mesh).
 !*
 !*    2. G_NUM is broadcast from the processor of rank NPES-1 to the rest
 !*       of the processors (0, 1, 2, ... , npes-2)
 !*
 !*    3. Each processor makes a copy of its own elements in the
 !*       array G_NUM_PP(NOD,NELS_PP). The local elements are those numbered
 !*       contiguously from IEL_START (the first element number on the
 !*       processor) to IEL_START + NELS_PP, where NELS_PP is the total
 !*       number of elements on the processor.
 !*
 !*    4. The global array G_NUM is deallocated to save space.
 !*  ARGUMENTS
 !*    INTENT(IN)
 !*
 !*    fname             : Character
 !*                        Filename. Any value accepted.
 !*                        Maximum character length not specified.
 !*
 !*    iel_start         : Integer
 !*                        The ID of the first finite element on the local
 !*                        processor (NUMPE)
 !*
 !*    ndim              : Integer
 !*                        Number of dimensions.
 !*
 !*    nels              : Integer
 !*                        Total number of elements in the mesh.
 !*
 !*    nn                : Integer
 !*                        Total number of nodes in the mesh.
 !*
 !*    numpe             : Integer
 !*                        The local processor ID or rank.
 !*
 !*    INTENT(INOUT)
 !*
 !*    g_num_pp(nod,nels) : Integer
 !*                        Global array with the nodal connectivity.
 !*
 !*  RESULT
 !*    Modifies the values of g_num_pp(nod,nels) in the calling program.
 !*  AUTHOR
 !*    Vendel Szeremi
 !*  CREATION DATE
 !*    02.01.2007
 !*  COPYRIGHT
 !*    (c) University of Manchester 2007-2010
```

```
!******
!*  Place remarks that should not be included in the documentation here.
!*
!*    There is currently no support for different element types in the
!*    same mesh. All finite elements in a mesh are assumed to be the same
!*    in this implementation.
!*
!*    An improvement would be to read and broadcast without creating the
!*    global array.
!*/

IMPLICIT NONE


INTEGER                   :: bufsize
                           ! size of buffer in broadcast

INTEGER                   :: iel
                           ! loop counter

INTEGER                   :: nod
                           ! nodes per element

INTEGER,INTENT(IN)        :: iel_start
                           ! ID of first element on local processor

INTEGER,INTENT(IN)        :: ndim
                           ! number of dimensions in problem

INTEGER,INTENT(IN)        :: nels
                           ! total number of elements in mesh

INTEGER,INTENT(IN)        :: nn
                           ! total number of nodes in mesh

INTEGER,INTENT(IN)        :: npes
                           ! number of processors in COMM_WORLD

INTEGER,INTENT(IN)        :: numpe
                           ! local processor ID or rank

INTEGER,INTENT(INOUT)     :: g_num_pp(:,:)
                           ! node numbers for each local element

INTEGER,ALLOCATABLE       :: g_num(:,:)
                           ! node numbers for all elements in the mesh

CHARACTER(*), INTENT(IN) :: fname
                           ! filename

!------------------------------------------------------------------------------
! 1. Allocate and populate global array G_NUM
!------------------------------------------------------------------------------

nod = UBOUND(g_num_pp,1)
ALLOCATE(g_num(nod,nels))

IF(numpe==npes)THEN
  OPEN(21,FILE=fname, STATUS='OLD', ACTION='READ', FORM='UNFORMATTED')
  READ(21) g_num
  CLOSE(21)
END IF

!------------------------------------------------------------------------------
! 2. Broadcast global array to all processors
!------------------------------------------------------------------------------

bufsize = UBOUND(g_num,1)*UBOUND(g_num,2)
CALL MPI_BCAST(g_num,bufsize,MPI_INTEGER,npes-1,MPI_COMM_WORLD,ier)

!------------------------------------------------------------------------------
! 3. Copy own data to local array and deallocate global array
```

```
 !-----------------------------------------------------------------------------

 g_num_pp = 0
 ielpe    = iel_start

 DO iel = 1,UBOUND(g_num_pp,2)
   g_num_pp(:,iel) = g_num(:,ielpe)
   ielpe           = ielpe + 1
 END DO

 DEALLOCATE(g_num)

 RETURN

END SUBROUTINE READALL_ELS_BINARY_FNAME
```