Dynamic Global Vegetation Model ORCHIDEE

Coding Guidelines

Main Authors	**Issuer**	
M. McGrath (IPSL/LSCE) J. Ryder (IPSL/LSCE) S. Luyssaert (IPSL/LSCE)	P. Peylin (IPSL/LSCE)	
	Date & Signature	
Contributors		
Working group: A. Cozic (IPSL/LSCE), Marie-Alice Foujols (IPSL), J. Ghattas (IPSL), F. Maignan (IPSL/LSCE), P. Maugis (IPSL/LSCE), J. Polcher (IPSL/LMD)		
Diffusion list		
orchidee-dev@ipsl.jussieu.fr		

Following version		
Index	**Date**	**Modifications**
1.0	2014/10/31	Creation

CONTENTS

1 Introduction

This is a collaborative document whose objective is to outline standard working procedures and coding style for ORCHIDEE.
This document is derived from the corresponding wiki page (http://forge.ipsl.jussieu.fr/orchidee/wiki/HowTo/FortranStandards) and is updated when needed.

2 Reference documents

Some climate models coding standards are referenced here:
http://www.easterbrook.ca/steve/2010/11/climate-model-coding-standards/
We particularly took inspiration from:

RD-1	NEMO coding conventions, NEMO System Team - C. Lévy, version 2, July 2010
RD-2	PRISM Coding Rules, Program for Integrated Earth System Modelling, May 2002

3 Generalities

3.1 English Language

Owing to the international user community, all naming of files, variables, modules, functions, and subroutines as well as all comments are to be written in English.
Note: copied from RD-2.

4 Style rules

The style rules are written to make it easier to read and understand the model. The main guideline is to write nice and tidy code. Think about the overview of the file: indent the code, add section numbers to the comments, align variable declarations and use capital letters to FORTRAN keywords, all this to have a more homogeneous code.

4.1 Argument list format

For function/subroutine calls, there should be the same number of arguments per line in the SUBROUTINE and in the corresponding CALL.
The arguments should if possible be aligned vertically as done in the example below.
We recommend using only five arguments per line (same as in RD-1 and RD-2).

```
CALL subroutine(arg1, arg2, arg3, arg4, arg5, &
                arg6, arg7, ...)
```

4.2 Sequence of arguments declaration

Related to point one, in the argument declaration of the subroutine, it's nice to have all the variables which are passed to/from:
- to be in the same order as they are listed,
- and respecting the order: INTENT(IN), then INTENT(INOUT), and last INTENT(OUT).

```
SUBROUTINE subroutine(arg1, arg2, arg3, arg4, arg5, &
                      arg6, arg7, ... )

    !
    !! 0. Variable and parameter declaration
    !

    !
    !! 0.1 Input variables
    !
    REAL(r_std), INTENT(in)                                 :: arg1         !! Time step (s)
    INTEGER(i_std), INTENT(in)                              :: arg2         !! Domain size (unitless)
    REAL(r_std), DIMENSION(kjpindex), INTENT(in)            :: arg3         !! Downwelling short wave flux

    !
    !! 0.2 Modified variables
    !
    INTEGER(i_std), INTENT(inout)                           :: arg4         !! … (unitless)
    REAL(r_std), INTENT(inout)                              :: arg5         !! … (s)
    REAL(r_std), DIMENSION(kjpindex), INTENT(inout)         :: arg6         !! …

    !
    !! 0.3 Output variables
    !
    INTEGER(i_std), INTENT(out)                             :: arg7         !! … (unitless)
```

Furthermore, we should try to give a standard logic to the order of arguments:

1) time information, 2) grid information, 3) I/O information, 4) physical INPUT variables, 5) physical IN/OUT variables, 6) OUTPUT variables.

The corresponding comment should be written on the same line after each variable declaration, this is mandatory for the documentation tool we use (Doxygen).

4.3 Comment at end of loop

For single loops and nested loops (loop within a loop) longer than about ten lines, it is helpful to repeat the loop instructions as a comment next to the END statement, as so:

```
eta_3_surf = 0.0d0

DO j = 1, nlevels

    DO k = j, 1
        jfactor = jfactor * (1.0d0 - jomega(k))

        ...
        ten more lines of code
        ...

    END DO ! k = j, 1

    eta_3_surf = eta_3_surf + (jomega_surf * jomega(j) * jfactor &
                               * sbsigma * temp_leaf_pres(j)**4.0d0)

END DO ! j = 1, nlevels
```

4.4 Naming conventions

4.4.1 Modules

Filenames have the extension .f90 (lowercase).
The file is named after the module it contains.
We use short names for the modules and all the routines inside a module have names of the type modulename_tasks_executed. So in stomate.f90 you can only find subroutines of the type stomate_*.

4.4.2 Counters

Limit to five characters. If the variable being looped over begins with "n", replace the "n" by an "i" for the counter name. For example: nvm → ivm, npts → ipts, nelements → ielem, naprts → ipart, ncirc → icirc, nleafage → ilage (the more logical ileaf is already used).

4.4.3 Variables and parameters

When possible, use the same name for call-parameters in the calling program and in the routine definition.
Avoid using a same identifier in different routines with different meanings or uses.
Choose names that really reflect the content or the use of the variable, placing yourself as an external reader trying to understand your code.

4.5 Formatting conventions

4.5.1 Capitalization

Always use capital letters for FORTRAN keywords: ALLOCATE, SUBROUTINE, DO, END DO, ... and intrinsic functions.

4.5.2 Use of space - indentation

Indentation of the code is very important to make it easier to read and understand the code. Always indent the code within conditional statements or loops, but don't use tabs, as the formatting will not be preserved across platforms.

Note: The emacs indent function works well for this, since it indents with spaces (even if you use the tab key).
Don't add a single comment character on an empty line, keep the line empty. Do not:

```
!
```

Similarly, don't use empty continuation lines containing only ampersand '&'.
Try to align the affectation signs '=' when doing multiple affectations on successive lines. Use spaces around it. Align also the '::' in declarations and chose one column at which to start by standard the trailing comments. Put one space after comas in argument lists (see example §4.2).
Starting a line with an ampersand is useless and should be avoided, except in case of a long string. The trailing '&' suffices to tell FORTRAN that the line carries on to next line.

4.5.3 Equations

Use brackets to improve readability (even though addition and subtraction are treated ahead of division and multiplication, it is easier for the reader to scan the equations if this is made explicit). Also, if the equation runs over several lines, try to break the expression at a close bracket or an addition/subtraction.
e.g. **a = (b * i) + (c / n)** is easier to read than **a = b * i + c / n**

Align multi-line equations according to the level of the operation. Aerate parenthesis content when made necessary by intense nesting:

```
  d1 = ( k_lin(ji+1,jst) / (avan(jst)*m*nvan(jst)) )        &
     * ( ((frac**(-un/m)) / (mc_lin(ji+1,jst) - mcr(jst)))  &
       + frac**(-un/m) -un ) ** (-m)
```

4.5.4 Line length

Although the maximum line length of FORTRAN 90 is 132 characters, try to keep your code to less than 100 characters per line - this preserves the formatting for those who work with small terminal windows on their computer and when producing a printout.
NOTE: If you are an emacs user, loading the column-marker.el file will help you highlight column 100 so you know where to terminate the line at.
There is an exception for the header and declaration parts of modules/subroutines/functions, whose length is 132 characters, due to detailed comments.

4.5.5 Special characters

Do not use non-ascii characters (typically, accentuated characters) in comments. They are non portable and can induce treatment failure by code analysis scripts.

5 Content rules

5.1 Constants

Never use "magic numbers", i.e. hard-coded numbers that cannot be traced to a literature formula. Externalization helps a lot with this. Similarly, when converting between units, use variables in constants.f90 for this purpose instead of using the number directly. This makes it more obvious exactly what one is doing.

5.2 Declaration for variables and constants

5.2.1 IMPLICIT NONE
Always add IMPLICIT **NONE** at the beginning of subroutine or module (the scope of such a module declaration extends to all procedures defined in the module).

5.2.2 INTENT
Always use the INTENT() attribute for arguments.

5.2.3 DIMENSION
Always use the DIMENSION statement in the declaration. This helps readability and allows the compiler to do consistency checks.

5.3 Reading of a parameter
When reading a parameter from the run.def file, use the getin (sequential mode)/getin_p (parallel mode) subroutine, see example below:

```
!Config Key   = VEGETATION_FILE
!Config Desc  = Name of file from which the vegetation map is to be
read
!Config If    = LAND_USE
!Config Def   = PFTmap.nc
!Config Help  = The name of the file to be opened to read a
vegetation
!Config         map (in pft) is to be given here.
!Config Units = [FILE]
!
filename = 'PFTmap.nc'
CALL getin_p('VEGETATION_FILE',filename)
```

5.4 Voluntary termination
In case the program is to be terminated, use the ipslerr subroutine and not a STOP statement. This enables to quit all processors properly.
See example below:

```
WRITE(numout,*) 'ERROR: We are thinning, but we have no trees left!'
WRITE(numout,*) 'ipts, ivm ',ipts, ivm
WRITE(numout,*) 'kill ',circ_class_kill_temp(:)
WRITE(numout,*) 'n ',circ_class_n_temp(:)
CALL ipslerr_p (3,'forestry', 'Thinning, but no trees left.',&
   'Look in the output file for ERROR.',&
   '')
```

The WRITE statements before ipslerr give a good amount of additional information (such as the pixel and PFT) to help solve the problem, and by writing to numout (instead of standard output, like ipslerr does) one knows which CPU is causing the problem. That can make it easier to isolate the pixel and create a small reproducer case.

5.5 Parallelization
The code is parallelized using both MPI and OpenMP. In coupled mode with LMDZ, the model can run in sequential, pure MPI and mixed MPI/OpenMP mode. In offline mode,

ORCHIDEE can run in sequential or pure MPI mode. The offline driver can also be compiled in mixed MPI/OpenMP mode but it can not run on more than 1 OMP thread.

Each development must be validated in the different parallelization modes. Running on different numbers of cores should give identical results for the same simulation setup, for example using 16MPI should give the same results as 8MPI*4OMP. The parallelization is transparent as far as there are no interactions between the grid cells. **Except if you develop the routing module**, interpolations or I/O issues such as reading or writing new files, you don't need to worry much about the parallelization. However, you have some rules to **respect:**

- Use the subroutines called restget_p, restput_p and getin_p instead of restget, restput and getin.
- Exception : if you restart a scalar, you have to use restget **followed by a bcast command**:

```
IF (is_root_prc) THEN
   var_name = 'day_counter'
   CALL restget (rest_id_stomate, var_name, 1   , 1     , 1, itime, &
            &                  .TRUE., xtmp)
   day_counter = xtmp(1)
   IF (day_counter == val_exp) day_counter = un
ENDIF
CALL bcast(day_counter)
```

For restput, follow the example:

```
IF (is_root_prc) THEN
   var_name = 'day_counter'
   xtmp(1) = day_counter
  CALL restput (rest_id_stomate, var_name, 1, 1, 1, itime, xtmp)
ENDIF
```

- Use histwrite_p instead of histwrite.
- Use ioconf_setatt_p instead of ioconf_setatt.
- Use ipslerr_p instead of ipslerr.
- For variables with the SAVE attribute, add a declaration

!$OMP THREADPRIVATE(name_var) as in the example below:

```
  REAL(r_std),ALLOCATABLE,SAVE,DIMENSION(:,:,:)  :: biomass
!! Biomass per ground area @tex $(gC m^{-2})$ @endtex
!$OMP THREADPRIVATE(biomass)
```

6 Documentation

6.1 Headers

The code has to be documented to benefit to all users. Here are header examples for module, subroutine and function. The specific signs (!, !!, !>\, !!\n and !_) are mandatory for the ‘Doxygen’ tool, which is used to automatically extract pdf and html documentations from the code.

6.1.1 Module header

Each module starts with a module header, see example below:

```
!
=============================================================================================================
====================
! MODULE       : forestry
!
! CONTACT      : orchidee-help _at_ ipsl.jussieu.fr
!
! LICENCE      : IPSL (2006)
! This software is governed by the CeCILL licence see ORCHIDEE/ORCHIDEE_CeCILL.LIC
!
!>\BRIEF       Gathers the main elements for forest management: the "forestry"
!! subroutine, which itself calls a set of subroutines
!! forestry_clear, clearcut, thinning, harvest, force_load, QsortC, and
!! Partition, and a set of functions used in these subroutines.
!!
!!\n DESCRIPTION: None
!!
!! RECENT CHANGE(S): None
!!
!! REFERENCE(S)    :
!! - Asael, S., 1999. Typologie des peuplements forestiers du massif vosgiens.
!! C.R.P.F. Lorraine-Alsace, Nancy, 54 p.
!! - Bellassen, V., Le Maire, G., Dhote, J.F., Viovy, N., Ciais, P., 2010.
!! Modeling forest management within a global vegetation model  Part 1:
!! model structure and general behaviour. Ecological Modelling 221, 24582474.
!! - Bellassen, V., Le Maire, G., Guin, O., Dhote, J.F., Viovy, N., Ciais, P.,
!! 2011a. Modeling forest management within a global vegetation model  Part 2:
!! model validation from tree to continental scale. Ecological Modelling 222,
```

```
!! 5775.
!!
!! SVN          :
!! $HeadURL: $
!! $Date: $
!! $Revision: $
!! \n
!_
================================================================================================================
====================
```

6.1.2 Subroutine header

Each subroutine starts with a subroutine header, see example below:

```
!!
================================================================================================================
====================
!! SUBROUTINE   : pheno_moigdd
!!
!>\BRIEF          The 'moigdd' onset model initiates leaf onset based on mixed temperature
!!               and moisture availability criteria.
!!               Currently PFTs 10 - 13 (C3 and C4 grass, and C3 and C4 agriculture)
!!               are assigned to this model.
!!
!! DESCRIPTION  : This onset model combines the GDD model (Chuine, 2000), as described for
!!               the 'humgdd' onset model (::pheno_humgdd), and the 'moi' model, in order
!!               to account for dependence on both temperature and moisture conditions in
!!               warmer climates. \n
!!               Leaf onset begins when the a PFT-dependent GDD threshold is reached.
!!               In addition there are temperature and moisture conditions.
!!               The temperature condition specifies that the monthly temperature has to be
!!               higher than a constant threshold (::t_always) OR
!!               the weekly temperature is higher than the monthly temperature.
!!               There has to be at least some moisture. The moisture condition
!!               is exactly the same as the 'moi' onset model (::pheno_moi), which has
!!               already been described. \n
!!               GDD is set to undef if beginning of the growing season detected.
!!               As in the ::pheno_humgdd model, the parameter ::t_always is defined as
```

```
!!                10 degrees C in this subroutine, as are the parameters ::moisture_avail_tree
!!                and ::moisture_avail_grass (set to 1 and 0.6 respectively), which are used
!!                in the moisture condition (see ::pheno_moi onset model description). \n
!!                The PFT-dependent GDD threshold (::gdd_crit) is calculated as in the onset
!!                model ::pheno_humgdd, using the equation:
!!                \latexonly
!!                \input{phenol_hummoigdd_gddcrit_eqn.tex}
!!                \endlatexonly
!!                \n
!!                where i and j are the grid cell and PFT respectively.
!!                The three GDDcrit parameters (::pheno_crit%gdd(j,*)) are set for each PFT in
!!                ::stomate_data, and three tables defining each of the three critical GDD
!!                parameters for each PFT is given in ::gdd_crit1_tab, ::gdd_crit2_tab and
!!                ::gdd_crit3_tab in ::stomate_constants. \n
!!                The ::pheno_moigdd subroutine is called in the subroutine ::phenology.
!!
!! RECENT CHANGE(S): None
!!
!! MAIN OUTPUT VARIABLE(S): ::begin_leaves - specifies whether leaf growth can start
!!
!! REFERENCE(S) :
!! - Botta, A., N. Viovy, P. Ciais, P. Friedlingstein and P. Monfray (2000),
!! A global prognostic scheme of leaf onset using satellite data,
!! Global Change Biology, 207, 337-347.
!! - Chuine, I (2000), A unified model for the budburst of trees, Journal of
!! Theoretical Biology, 207, 337-347.
!! - Krinner, G., N. Viovy, N. de Noblet-Ducoudre, J. Ogee, J. Polcher, P.
!! Friedlingstein, P. Ciais, S. Sitch and I.C. Prentice (2005), A dynamic global
!! vegetation model for studies of the coupled atmosphere-biosphere system, Global
!! Biogeochemical Cycles, 19, doi:10.1029/2003GB002199.
!!
!! FLOWCHART    :
!! \latexonly
!! \includegraphics[scale = 1]{pheno_moigdd.png}
!! \endlatexonly
!! \n
!_
================================================================================================================
==================
```

6.1.3 Function header

Each function starts with a function header, see example below:

```
!!
================================================================================================================
==================
!! FUNCTION     : bm_vol
!!
!>\BRIEF        ! This allometric function computes biomass as a function of
!! volume at stand scale. biomass \f$gC m^{-2}) = f(volume (m^3 ha^{-1}))\f$
!!
!! DESCRIPTION : None
!!
!! RECENT CHANGE(S): None
!!
!! RETURN VALUE : bm_vol
!!
!! REFERENCE(S) : See above, module description.
!!
!! FLOWCHART    : None
!! \n
!_
================================================================================================================
==================
```

6.2 Equations and graphics

Equations may be saved as separate latex files. These files are stored in one of the sub folders in ORCHIDE/DOC/ORCHIDEE directory. An equation is written in latex contained between \begin{equation} and \end{equation}. The file is named after the subroutine/function it belongs to.

Examples:
!!\latexonly
!!\input{interception.tex}
!!\endlatexonly

!!\latexonly
!!\includegraphics[scale = 1]{choisnelvariables.pdf}
!!\endlatexonly

6.3 Table of content

The 'Doxygen' tool extracts comments preceded by a number e.g. `!! 1. Initialize variables` and organizes these comments as a 'Table of content'. Take this in mind when numbering sections of code and use short headers. More information can be given below the header.

6.4 Declaration part

For the declaration part, please use:

```
!! 0. Variables and parameter declaration
!! 0.1 Input variables
!! 0.2 Modified variables
!! 0.3 Output variables
!! 0.4 Local variables
```

6.5 Variables

Variables are commented AFTER variable declaration.

6.6 Document units

Use comment lines to document units and unit conversions.
A variable without units is labeled as `(unitless)`.
If the range of the variable is known, this range is given i.e. `(0-1, unitless)`.
Units are written in latex and no fractions should be used i.e. `(m s^{-1})` instead of `(m/s)`.
So special superscript characters like ^{2} or ^{3} are banned.
Units are contained within the following tags `@tex ($m s^{-1}$) @endtex`.
This latex formatting is used to get nice outputs in the pdf documentation generated with the Doxygen tool.

6.7 Document your thinking

Document your thinking rather than simply describing the FORTRAN code. In particular, if you tried a couple of ways to structure the code before deciding on what you leave in, mention what you tried and why that didn't work in order to prevent people in the future from redoing those same tests.
Suppress obsolete or old comment markers dedicated only to you or a small set of contributors, aiming at precising a specific modification that needed to be pointed out at a

given time of the developments. Prefer documenting the header, the svn commit, or your own development booknote, since the modifications are traced by svn.

7 Code structure

7.1 Modules need to be self sufficient

They have their own prognostic variables for which they need to manage the allocation, restart, ... Prognostic variables are not to be exchanged with other modules (i.e. private to the module) else you cannot change one module without affecting the others.

7.2 Modules have at least 3 subroutines

1) module_main: manages the actions to be taken and the calling sequence,
2) module_init: initialises the module (configuration, restart, allocation),
3) module_clear: deallocates the internal memory.
Currently in most cases only module_main and module_clear are public.
The first call to module_main triggers only the module initialization phase under a boolean flag like 'firstcall_module'. This initialization can be done either in the _main or in a dedicated routine called by the main but should not be performed outside them. This init phase does not perform any non-initialization computation as the input variables are not all guaranteed to be valid. For robustness, setting " firstcall_module = .FALSE" should be done at the end of the initialization phase in the top-routine that tested the flag first (i.e. most of the time in the _main() routine). Complete initialisation (including allocation) must be performed in a single pass. For example, and even if several init. flags can be used, only one initialization "RETURN" should close the initialization phase. Deallocation is forbidden there and must be performed in the _clear() routine only. It is only the second call to module_main that will start the onward calculations.
Note that all modules do not need a main subroutine. This is often the case for modules dealing with technical issues like I/O, parallelism, etc.

7.3 Circular structure call

In the calling tree, subroutines should not call higher level routines (i.e. one that called the current routine directly or indirectly): circularity is forbidden.

8 Debugging and optimisation

8.1 Control text output

8.1.1 PRINTLEV: global parameter

PRINTLEV is an externalised parameter used to define a global level of output text information. Default value is 1.

PRINTLEV definition:

0 no output.
1 minimum writing for long simulations only at initialization and finalization phase (default)
2 more basic information for long simulations, some daily information can be written, nothing must be written at each time step.
3 first debug level: entering and leaving subroutines can be reported.

4 higher debug level: input parameters to major subroutines can be reported, other debug information.

In the code, use the following syntax:
IF (printlev>=1) WRITE(numout,*) 'This is a very important write statement...'
IF (printlev>=3) WRITE(numout,*) 'This is a debug print...'

8.1.2 PRINTLEV_modulename: local parameter to one module

The function get_ printlev('modulename') makes it possible to have a local write level in a module. Setting PRINTLEV_modulename in run.def changes then the write level in that module. The default value is the global PRINTLEV value.

This functionality is available only in modules where the function get_printlev() is called. get_printlev should be called once in the initialization part for the module to define a new local saved variable. This new variable should be used in the whole module for all write statements instead of printlev. For example:

In module sechiba:
INTEGER, SAVE :: printlev_loc

In subroutine init_sechiba:
printlev_loc=get_printlev('sechiba') => this function will read the variable PRINTLEV _sechiba from run.def.

In the rest of the module, all write statements should be done as follows:
IF (printlev_loc>=1) WRITE(numout,*) 'This is a low level print'
IF (printlev_loc>=3) WRITE(numout,*) 'This is a debug print'

8.2 Don't forget the ELSE

If you are using an IF...ELSEIF....ENDIF loop, it is recommended that you include an ELSE statement at the end to catch any situation not covered in the other cases. Too many bugs are found because an IF statement is not triggered due to something the programmer didn't think of. This is especially problematic when the programmer thinks to him/herself, "This value will always be in this range, so I don't have to consider other possibilities"...and then one day things change.
If the ELSE statement doesn't do anything, it could have something written in a comment instead just so that other people know that nothing needs to be done in some cases. Even better would be a call to ipslerr to stop the code and say we should not have been here.

```
IF () THEN
  ! do something
  blah
ELSEIF () THEN
  ! do something else
  blah blah
ELSE
  ! do something, or not, but at least you should be aware of the
possibility
ENDIF
```

There are few cases when this is not mandatory such as:

```
IF (firstcall)THEN
  ! do something
  blah
ENDIF
```

8.3 Select default cases last

If you are using a SELECT CASE ... CASE ... END SELECT statement, it is recommended for the same reason that you include a “CASE default” statement at the end. Despite the fact that FORTRAN 90 allows you to place it anywhere in the sequence, it is mandatory to place it last to allow proper code analysis for assimilation.

```
SELECT CASE (var)
CASE (val1)
  ! do something if var == val1
  blah
CASE (val2)
  ! do something else
  blah blah
CASE default
  ! other cases; note the absence of parenthesis
END SELECT
```

8.4 Loop ordering

Be mindful of loop ordering for best memory access (performance). If the embedded loops are independent, then the first index should correspond to the most inner loop. In other words:

```
DO k=1,nk
  DO j=1,nj
    DO i=1,ni
      f(i,j,k) = ...
    ENDDO
  ENDDO
ENDDO
```

This ensures that one accesses contiguous memory blocks during the loop, which makes it faster.

8.5 Portability

Test your modifications on several computers using different compilers. Activate compile debug options for testing the code. In ORCHIDEE/arch directory, you find compile debug options for different computers. Get more information on the ORCHIDEE wiki.