



MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m12e/2025/12.11.12.55-NTC

**DOCUMENTO TÉCNICO NORMATIVO (DTN)
PADRÃO DE CODIFICAÇÃO PARA O MONAN
(MODEL FOR OCEAN-LAND-ATMOSPHERE
PREDICTION)**

**DTN-01 - VERSÃO 0.1.0
GCC - GRUPO DE COMPUTAÇÃO CIENTÍFICA
DIMNT - DIVISÃO DE MODELAGEM NUMÉRICA DO SISTEMA
TERRESTRE
INPE - INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS**

Eduardo Georges Khamis
Carlos Renato de Souza
Denis Magalhães de Almeida Eiras
João Messias Alves da Silva
Marcelo Paiva Ramos
Kleucio Claudio
Luiz Flávio Rodrigues

URL do documento original:
<<http://urlib.net/8JMKD2USNRW34T/4EP3N68>>

INPE
São José dos Campos
2025

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Coordenação de Ensino, Pesquisa e Extensão (COEPE)
Divisão de Biblioteca (DIBIB)
CEP 12.227-010
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/7348
E-mail: pubtc@inpe.br

CONSELHO DE EDITORAÇÃO E PRESERVAÇÃO DA PRODUÇÃO INTELLECTUAL DO INPE - CEPPII (PORTARIA Nº 176/2018/SEI-INPE):

Presidente:

Dr. Thales Sehn Korting - Coordenação-Geral de Ciências da Terra (CGCT)

Membros:

Dr. Antonio Fernando Bertachini de Almeida Prado - Conselho de Pós-Graduação (CPG)

Dr. Evandro Marconi Rocco - Coordenação-Geral de Engenharia, Tecnologia e Ciência Espaciais (CGCE)

Dr. Heyder Hey - Coordenação-Geral de Infraestrutura e Pesquisas Aplicadas (CGIP)

Simone Angélica Del Ducca Barbedo - Divisão de Biblioteca (DIBIB)

BIBLIOTECA DIGITAL:

Dr. Gerald Jean Francis Banon

Clayton Martins Pereira - Divisão de Biblioteca (DIBIB)

REVISÃO E NORMALIZAÇÃO DOCUMENTÁRIA:

Simone Angélica Del Ducca Barbedo - Divisão de Biblioteca (DIBIB)

André Luis Dias Fernandes - Divisão de Biblioteca (DIBIB)

EDITORAÇÃO ELETRÔNICA:

Ivone Martins - Divisão de Biblioteca (DIBIB)

André Luis Dias Fernandes - Divisão de Biblioteca (DIBIB)



MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m12e/2025/12.11.12.55-NTC

**DOCUMENTO TÉCNICO NORMATIVO (DTN)
PADRÃO DE CODIFICAÇÃO PARA O MONAN
(MODEL FOR OCEAN-LAND-ATMOSPHERE
PREDICTION)**

**DTN-01 - VERSÃO 0.1.0
GCC - GRUPO DE COMPUTAÇÃO CIENTÍFICA
DIMNT - DIVISÃO DE MODELAGEM NUMÉRICA DO SISTEMA
TERRESTRE
INPE - INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS**

Eduardo Georges Khamis
Carlos Renato de Souza
Denis Magalhães de Almeida Eiras
João Messias Alves da Silva
Marcelo Paiva Ramos
Kleucio Claudio
Luiz Flávio Rodrigues

URL do documento original:
<<http://urlib.net/8JMKD2USNRW34T/4EP3N68>>

INPE
São José dos Campos
2025



Esta obra foi licenciada sob uma Licença Creative Commons Atribuição-NãoComercial 3.0 Não Adaptada.

This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License.

Histórico de Revisões

Data	Versão	Descrição	Autor(es)
08/07/2025	0.1.0	Criação do documento, versão inicial.	Todos

RESUMO

Este Documento Técnico Normativo (DTN) tem como objetivo estabelecer um conjunto padronizado de regras e boas práticas para a escrita de código no contexto do MONAN (Model for Ocean laNd and Atmosphere predictioN), visando à melhoria da manutenibilidade e da qualidade do modelo sob a perspectiva da engenharia de software. O documento classifica as diretrizes em dois tipos: regras mandatórias, que são obrigatórias e devem ser seguidas por todos os desenvolvedores; e regras recomendadas, que devem ser adotadas sempre que possível, podendo ser flexibilizadas em situações justificadas. A adoção dessas convenções contribui para a padronização, consistência e sustentabilidade do desenvolvimento do modelo ao longo do tempo.

Palavras-chaves: padrões de codificação; boas práticas de programação; qualidade de software; manutenção de código; MONAN; documento técnico normativo.

LISTA DE SIGLAS E ABREVIATURAS

DTN	Documento Técnico Normativo
FORD	FORtran Documenter
FORTRAN	FORmula TRANslation System
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
IDE	Integrated Development Environment
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
MONAN	Model for Ocean-laNd-Atmosphere Prediction
MPAS	Model for Prediction Across Scales

SUMÁRIO

1. Introdução	5
2. Refatoração	7
3. Padrão de texto utilizados neste documento	7
4. Regras Mandatórias de Codificação Padrão	8
4.1. Conformidade com o Padrão Fortran 90/95 (CF)	8
4.2. Codificação (CO)	9
4.3. Documentação (DO)	10
4.4. Entrada/Saída - I/O (ES)	10
4.5. Formatação de Código (FC)	10
4.6. Modularidade e Reuso (MR)	10
5. Regras Recomendadas de Codificação Padrão	11
5.1. Conformidade com o Padrão Fortran 90/95 (CF)	11
5.2. Codificação (CO)	11
5.3. Documentação (DO)	21
5.4. Entrada/Saída - I/O (ES)	26
5.5. Formatação de Código (FC)	27
5.6. Modularidade e Reuso (MR)	32
Referências bibliográficas	37
Referências consultadas	39

1. Introdução

É sabido que a qualidade do produto de *software* é um objetivo do processo de desenvolvimento.

A qualidade de um *software* é composta por 3 partes, segundo a norma ISO/IEC 9126 (2001), (i) qualidade interna, (ii) qualidade externa e (iii) qualidade em uso. Este Documento Técnico Normativo (DTN-01) trata de algumas qualidades internas e externas e de especificar normas de codificação para um dos componentes de um modelo de qualidade: o produto. De forma mais específica, esse documento relaciona regras obrigatórias e desejáveis a serem aplicadas durante a codificação, isto é, na escrita do código.

Em todas as qualidades internas e externas, a conformidade (capacidade do produto de *software* de estar de acordo com normas, convenções e leis) é de fundamental importância para garantir a qualidade do que se produz. A maturidade de um *software* só pode ser garantida se uma série de requisitos são atendidos. Dentre os requisitos, há um que garante o futuro do *software* e de um desenvolvimento contínuo num contexto de modelo de desenvolvimento de *software*: a manutenibilidade.

Segundo a norma ISO/IEC 9126 (2001), revisada pela norma ISO/IEC 25010 (2011), a manutibilidade é a capacidade (ou facilidade) do produto de *software* ser modificado, incluindo tanto as melhorias ou extensões de funcionalidade quanto às correções de defeitos, falhas ou erros. Em outras palavras, a capacidade de um *software* de comportar modificações, melhorias, correções, ou adaptações a novos requisitos. A manutenibilidade é dividida em cinco subcaracterísticas:

- Analisabilidade: É fácil de encontrar uma falha, quando ocorre?
- Modificabilidade: É fácil modificar e adaptar?
- Modularidade: Existe independência funcional entre os módulos do programa?
- Testabilidade: É fácil validar o software modificado?
- Reusabilidade: É fácil reutilizar o software em outras aplicações?

Analisar, modificar e reusar um *software* requer que se conheça seu funcionamento e que haja uma padronização clara, compreensível em sua escrita. Dessa forma, atuar sobre o

produto fica mais simples, mais rápido e evita a introdução de erros (*bugs*) que comprometem o funcionamento da solução.

A padronização de codificação faz parte do ciclo de vida do desenvolvimento de *software*, que está associada à etapa de projeto do *software*, a qual tem também os objetivos de definir a arquitetura, linguagem, ferramentas, *frameworks* e plataformas (CARULLO, 2020).

“Os padrões de codificação são os procedimentos que definem uma linguagem de programação específica. Eles ajudam a especificar o estilo, os métodos e os procedimentos. (...) Melhorar a qualidade do código traz uma série de vantagens principais. Garantir que a qualidade do código seja boa pode aumentar a eficiência do seu projeto e reduzir o risco de falha do seu projeto. Um código de boa qualidade também é simples, o que significa que os níveis de complexidade são reduzidos e fáceis de manter. No geral, tudo isso resulta em projetos e software mais bem-sucedidos e econômicos.”
(BRING, 2024)

O projeto GNU (2020) possui um conjunto de regras de codificação, o GNU Coding Standards e cita que espera-se com a aplicação delas manter o sistema limpo, consistente e fácil de instalar e ainda que o documento possa ser interpretado como um guia para escrever programas portáteis, robustos e confiáveis. As regras de codificação do compilador GCC (2024) engloba as regras do GNU (2020) e suas próprias regras que devem ser seguidas.

O projeto MPAS (2013) no documento *MPAS Developers Guide* apresenta também um conjunto de regras para codificação com o objetivo de manter o código mais uniforme possível em todo o sistema.

Dessa forma o *Model for Ocean laNd and Atmosphere predictioN* (MONAN) deve ter seu código escrito dentro de um padrão comum, auditável e sua manutenção deve ser facilitada por uma codificação limpa e bem documentada.

2. Refatoração

Todos os *softwares* importados ou suas partes, isto é, *softwares* não desenvolvidos pela equipe de desenvolvimento do MONAN e que serão acrescentados ao modelo, deverão passar por uma refatoração de código de forma a garantir que o padrão de codificação destes pacotes estejam em conformidade com as regras estabelecidas neste documento. Existem ferramentas e ambientes de desenvolvimento (Integrated Development Environment - IDEs) que auxiliam no trabalho de refatoração, por exemplo, *Eclipse Photran* (2024) e podem auxiliar sobremaneira a tarefa de refatoração. Além disso é saudável que todos os desenvolvedores envolvidos utilizem ferramentas modernas de codificação e escrita, como o *Visual Studio Code* (VSCODE, 2024), *Sublime Text* (SUBLIME, 2024) ou *Atom* (2024) configuráveis e programáveis, que facilitem tanto a edição dentro das regras, quanto a refatoração de código. Também é recomendável que sejam desenvolvidas pelos participantes do projeto, preferencialmente com parcerias internacionais, ferramentas que facilitem a refatoração de códigos.

Parte da padronização usada neste documento tem como referência os tópicos baseados no padrão *Fortran 90/95 Coding Conventions* (FORTRAN, 2024).

3. Padrão de texto utilizados neste documento

Esse documento usa dois tipos de indicadores de regras. No capítulo 4 as regras mandatórias, ou seja, regras obrigatórias que devem ser seguidas pelos desenvolvedores e, no capítulo 5, as regras recomendadas, isto é, regras que podem ser rejeitadas em casos especiais onde as mesmas não podem ser cumpridas pelos desenvolvedores. Todos os trechos de código exemplo mostrados neste documento serão mostrados em modo "code blocks" conforme o exemplo abaixo:

```
logical function CheckAllInputs(in_val)
  real, intent(in) :: in_val
  return .true.
end function CheckAllInputs
```

As regras são agrupadas respeitando a seguinte nomenclatura: **TR.GR.keyword**, onde temos:

- **TR** para o “Tipo da Regra”, opções: M (Mandatária) ou R (Recomendada).
- **GR** para o “Grupo da Regra”, opções: CF (Conformidade com o Padrão Fortran 90/95), CO (Codificação), DO (Documentação), ES (Entrada/Saída - I/O), FC (Formatação de Código) e MR (Modularidade e Reuso).
- **keyword** para a “Palavra-Chave” da regra deve ser uma ou mais palavras, no formato `snake_case`, que representam unicamente uma regra dentro de TR e GR

4. Regras Mandatórias de Codificação Padrão

4.1. Conformidade com o Padrão Fortran 90/95 (CF)

M.CF.comandos_obsoletos - É proibido o uso de comandos obsoletos para Fortran 90/95 como os listados abaixo:

- *common*: Deve ser substituído por módulos (memória). Seu uso como o equivalence é fonte de invasão de memória;
- *data*: Não deve ser usado. As variáveis devem ser inicializadas por subrotinas de inicialização em cada módulo. Em casos específicos de constantes usar parameter.
- *double precision*: utilize sempre as expressões com Kind tanto para reais quanto para inteiros. Esta prática permite controle explícito sobre a precisão, bem como agrega portabilidade e segurança ao código.
- *equivalence*: Proibido por ser fonte de invasão de memória;
- *pause*: Proibido devido a máquinas que usam submissão de jobs não o executarem sendo fonte de problemas de portabilidade;
- *save*: Deve ser substituído por variáveis em módulos. Seu uso torna o código não thread safe e não puro impedindo os recursos de OpenMP;

M.CF.palavras_reservadas - As palavras reservadas da linguagem devem ser todas declaradas com letras minúsculas. Palavras-chave Fortran (por exemplo, data) não devem ser usadas como nomes de variáveis. São palavras reservadas (104):

abstract, allocatable, allocate, assign, associate, asynchronous, backspace, bind, block, block data, call, case, class, close, codimension, common, concurrent, contains, contiguous,

continue, critical, cycle, data, deallocate, deferred, dimension, do, elemental, else, elseif, elsewhere, end, endfile, endif, entry, enum, enumerator, equivalence, error, exit, extends, external, final, flush, forall, format, function, generic, goto, if, implicit, import, include, inquire, intent, interface, intrinsic, lock, memory, module, namelist, non_overridable, nopass, nullify, only, open, operator, optional, parameter, pass, pause, pointer, print, private, procedure, program, protected, public, pure, read, recursive, result, return, rewind, rewrite, save, select, sequence, stop, submodule, subroutine, sync, sync all, sync images, target, then, unlock, use, value, volatile, wait, where, while, write.

4.2. Codificação (CO)

M.CO.implicit_none - Todos os programas, módulos ou *procedures* devem possuir a declaração de *implicit none* no início. O não uso de *implicit none* implica em tipos de variáveis assumidos pelo compilador que pode gerar erros difíceis de serem detectados, assim ao usá-lo o programador passa a ter controle sobre todas as declarações de variáveis.

M.CO.variáveis_inicialização - Inicialize todas as variáveis (com exceção dos ponteiros, parâmetros, constantes). Não assuma atribuições de valor padrão da máquina.

M.CO.intent - O atributo *intent* deve ser usado para todos argumentos em uma função ou subrotina, com exceção de ponteiros e estruturas (tipos derivados) que não são definidos pelo padrão. Deixa clara a intenção do argumento dentro da subrotina. Os *intents* podem ser “in”, “out” ou “inout”. As variáveis de entrada e saída devem estar declaradas com o *intent* adequado (“in”, “inout” ou “out”).

```
function Test(var_a, var_b) result(compar)
  implicit none
  real, intent(in) :: var_a
  !!Primeiro real da comparação
  real, intent(in) :: var_b
  !!Segundo real da comparação

  logical :: compar

  compar = (var_a == var_b)

end function Test
```

M.CO.arrays_automáticos - O mecanismo preferencial para alocação dinâmica de memória são os arrays automáticos (passados por parâmetro) em oposição aos *arrays "allocatable"* ou *"pointer"* para os quais a memória deve ser explicitamente alocada e desalocada; o espaço alocado usando *"allocatable"* ou *"pointer"* deve ser explicitamente liberado usando a instrução *"deallocate"*.

4.3. Documentação (DO)

Sem regras mandatórias nesta categoria.

4.4. Entrada/Saída - I/O (ES)

Sem regras mandatórias nesta categoria.

4.5. Formatação de Código (FC)

M.FC.snake_case - Nomes de variáveis compostos com mais de uma palavra devem usar o estilo *snake_case*, isto é, as palavras em minúsculo e as demais separadas por sub traços (*underscore*). Exemplo:

```
integer :: count_particles
!! Number of particles count [#]
real, allocatable :: aer_mass_cape(:, :, :)
!! Mass of aerosol on cape waves [g/m^3]
```

M.FC.indentação - Como regra geral para construção de fontes, definiu-se a indentação com 4 (quatro) espaços. Não usar tabulação para esse fim, pois cada editor pode abrir a indentação com *tabs* de forma errada. Definir uma indentação deixa o código mais claro. Tabulações não fazem parte do conjunto de caracteres FORTRAN.

4.6. Modularidade e Reuso (MR)

Sem regras mandatórias nesta categoria.

5. Regras Recomendadas de Codificação Padrão

5.1. Conformidade com o Padrão Fortran 90/95 (CF)

R.CF.conformidade - O código-fonte deve estar em conformidade com o padrão ISO/IEC Fortran 95 (ISO/IEC 1539:1997).

R.CF.kind - Precisão: as parametrizações não devem depender de sinalizadores de fornecedor para definir uma precisão de ponto flutuante padrão ou tamanho inteiro. O recurso F90/95 KIND deve ser usado em seu lugar, assim permitindo controle explícito sobre a precisão, bem como agregando portabilidade e segurança

5.2. Codificação (CO)

R.CO.extensão_de_arquivo_portabilidade - Nenhuma extensão dependente de compilador ou plataforma deve ser usada. Isto é, o código desenvolvido não pode fazer uso de funções intrínsecas de uma única versão de compilador e deve estar disponível em padrão Fortran ISO. Esta prática trará portabilidade ao código.

R.CO.especificadores_erro_IO - Nenhum uso deve ser feito de valores especificadores de erro dependentes do compilador (por exemplo, valores IOSTAT ou STAT).

R.CO.gfortran_compatível - O código-fonte deve ser compilado e executado obrigatoriamente em gfortran que faz parte da GNU *Compiler Collection*.

§1. Essa regra não impede que o mesmo seja compilado em outros compiladores. De fato deve ser compilado no maior número possível deles, garantindo a portabilidade.

R.CO.f90 - Todos os arquivos de código em Fortran terão sua extensão .F90 para não haver necessidade de distinguir códigos com diretivas de pré-processamento de outros.

R.CO.stop - Todos os comandos *stop* são proibidos. Para isso deve ser criada uma função de *StopExecution* que realize o encerramento do programa. Somente nela é permitido o uso de um *stop*.

§1. A função *StopExecution* deve exibir uma mensagem e o ponto de parada (*procedure* e arquivo fonte).

§2. Em caso de execuções paralelas a função *stop* deve encerrar a seção paralela do código para evitar múltiplos *prints* da mensagem de parada.

```
! -----  
function StopExecution(mensagem, fonte, rotina, processor) result(fim)  
  !! ## Para uma execução e imprime uma mensagem  
  !!  
  !! Author: Rodrigues, L.F. [LFR]  
  !!  
  !! e-mail: <mailto:luiz.rodrigues@inpe.br>  
  !!  
  !! Date: 13Outubro2022 11:03  
  !!  
  !! -  
  !! **Full description**:  
  !!  
  !! Interrompe uma execução e imprime uma mensagem  
  !!  
  !! -  
  use mpi  
  
  implicit none  
  ! Parameters:  
  character(len=*), parameter :: p_procedure_name = 'StopExecution'  
  
  ! Variables (input):  
  character(len=*), intent(in) :: mensagem  
  !! Mensagem a ser impressa  
  character(len=*), intent(in) :: fonte  
  !! Nome do arquivo fonte  
  character(len=*), intent(in) :: rotina  
  !! Nome do procedure  
  integer, intent(in), optional :: processor  
  !! Se é paralelizada o número do processador deve estar presente  
  
  ! Local variables:  
  integer :: ierr  
  integer :: fim  
  
  fim = 0  
  if(present(processor)) then  
    call MPI_Barrier(MPI_COMM_WORLD, ierr)  
    if(processor == 0) then  
      print*, '#####'  
      print *, "Fortran Stop - "//fonte//": "//rotina//"()  
      print *, mensagem  
      print , '#####'  
      print *, ''  
    end if  
  end if  
end function
```

```

        call mpi_finalize(ierr)

    endif
else
    print *, '#####'
    print *, "Fortran Stop - "//fonte// : "//rotina//()"
    print *, mensagem
    print *, '#####'
    print *, ''
    stop
endif
end function StopExecution

```

R.CO.laços_colapsado - Desvios condicionais ou laços com blocos “if/end if” e “do/end do” devem usar o “end” não colapsado, isto é, usar “end if” e “end do” ao invés de “endif” e “enddo”.

```

outer: do i_Count1 = 1,10
    do i_Count2 = 1,10
        if ( i_Count1 == 2 .and. i_Count2 == 3 ) cycle outer
        z(i_Count2, i_Count1) = 1.0d0
        if(i_Count1 > 7) then
            Print *, 'Just for test: ', i_Count2
            call adjustCount1Value(i_Count1)
        end if
    end do
end do outer

```

R.CO.constantes_parameter - Constantes precisam ser definidas com o atributo *parameter*. Para declarações utilize sempre uma constante por linha para facilitar a documentação e entendimento.

§1. Toda constante física deve iniciar com uma letra “c” seguida de um sublinhado “_”.

§2. Sempre coloque um comentário na linha seguinte, precedido por duas exclamações (“!!”), explicando a constante e sua unidade.

```

real, parameter :: c_kb = 1.3806504e-23
!! boltzmann constant [jk-1]

```

R.CO.constantes_não_físicas - Todas as constantes não físicas (*parameters*), precisam ser definidas com o atributo *parameter*. Para declarações, utilize sempre uma constante por linha para facilitar a documentação e entendimento.

§1. Toda constante não física deve iniciar com uma letra “p” seguida de um sublinhado “_”.

§2. Sempre coloque um comentário na linha seguinte, precedido por duas exclamações (“!!”), explicando o parâmetro.

```
! Parameters for dump functions
integer, parameter :: p_pi = 3.1415926
!! The pi value
integer, parameter :: p_notice = 1
!! Just a notice
integer, parameter :: p_warning = 2
!! Just a warning
integer, parameter :: p_fatal = 3
!! This is a Fatal message
integer, parameter :: p_kill = 4
!! Kill the model Signal
integer, parameter :: p_continue = 5
!! Continue run Signal
logical, parameter :: p_yes = .true.
!! Yes is the .true. value
logical, parameter :: p_no = .false.
!! No is the false value
```

R.CO.números_argumentos - Números embutidos em código devem ser evitados quando passados por listas de argumentos, pois um sinalizador de compilador, que define uma precisão padrão para constantes, não pode ser garantido. Use *parameters* para defini-los.

R.CO.variáveis_inicialização_tipo - Não inicializar variáveis de um tipo com valores de outro tipo. Esta prática contribui para a segurança dos resultados numéricos, bem como pode evitar erros difíceis de serem detectados, pois o fortran faz conversões implícitas de dados que podem levar a perda de precisão numérica e resultados não esperados.

R.CO.variáveis_inicialização_valor - Não inicializar uma variável com algum valor na sua declaração, exceto se for usado “parameter” (usada para inicializar constantes). As

inicializações devem ser realizadas em funções próprias para esse fim. Ao inicializar uma variável com valor, a depender do tamanho do código, ocorreria perda de manutenibilidade, caso tenha necessidade de modificá-lo em vários locais.

R.CO.indice_array_inteiro - As expressões subscriptas em matrizes e arrays devem ser apenas do tipo inteiro.

R.CO.arrays_alocáveis - Quando possível o uso de arrays alocáveis é preferível ao uso de ponteiros. Isto minimiza os riscos de vazamentos de memória e fragmentação de *heap*.

R.CO.variáveis_ponteiro_null - Sempre inicialize variáveis de ponteiro em sua instrução de declaração usando o intrínseco `null()` . Ex: `integer, pointer :: x => null()`

R.CO.alocação_memória - Em uma determinada unidade de programa, não aloque repetidamente um mesmo espaço de memória, desalocar e em seguida alocar um bloco maior de espaço gera grandes quantidades de memória inutilizável.

R.CO.alocação_memória_check - Ao alocar memória para uma variável, verificar se esta já não foi previamente alocada:

```
if (.not. allocated(x)) allocate(x(i))
```

obs: quando a alocação for a um ponteiro deve-se verificar se o mesmo já se encontra associado:

```
if (.not. associated(x)) allocate(x(i))
```

R.CO.desalocação_memória_check - Ao desalocar memória de uma variável, verificar se está mesmo alocada:

```
if (allocated(x)) deallocate(x)
```

R.CO.operandos_precisão_numérica - Não use os operadores `==` e `/=` com expressões de ponto flutuante como operandos. Em vez disso, verifique o desvio da diferença de um limite de precisão numérica predefinido (por exemplo, comparação de epsilon - precisão do tipo).

§1. Defina um epsilon mínimo de acordo com a precisão da máquina. Para isso use a função intrínseca *tiny(x)*, sendo *x* do tipo que deseja comparar. Compare se a diferença está dentro do intervalo de epsilon.

R.CO.conversões_tipo - Em expressões e atribuições de modo misto (onde variáveis de tipos diferentes são misturadas), as conversões de tipo devem ser escritas explicitamente (não assumidas). Não compare expressões de tipos diferentes, por exemplo. Execute primeiro e explicitamente a conversão de tipo.

R.CO.assumed_shape - O uso de *“assumed shape”* é recomendado ao passar vetores/matrizes para funções e subrotinas.

R.CO.variáveis_não_usadas - Remova as variáveis não utilizadas.

R.CO.código_debug - Remova o código que foi usado para depuração (*debug*) quando a tarefa para isso for concluída.

R.CO.parenteses_ordem - Use parênteses o tempo todo para controlar a ordem de avaliação nas expressões. Não se baseie na ordem de precedência em expressões longas e de difícil compreensão.

R.CO.arrays_alocáveis - Com arrays automáticos, potencialmente grandes, procure utilizar *arrays* alocáveis. Deste modo, caso não haja espaço suficiente durante a alocação do *array* na pilha (*stack*), o programa não irá falhar.

R.CO.arrays_desalocação - Ao utilizar *arrays* alocáveis, prefira os desalocar (*deallocate*) explicitamente quando não forem mais necessários.

R.CO.arrays_alocáveis_ponteiros - É altamente recomendada a utilização de *arrays* alocáveis em vez de ponteiros. *Arrays* alocáveis são agora permitidos como componentes de estruturas (*types*) nas extensões modernas do Fortran. Deste modo, a utilização de ponteiros não se faz tão necessária, uma vez que são demasiadamente lentos e seus

argumentos *dummy* são impraticáveis. Estes argumentos não são nada práticos, visto que os argumentos reais necessitam sempre ter o ponteiro ou o atributo de destino (ponteiros em C são o oposto).

R.CO.laços_grandes - Laços muito grandes devem ser evitados.

§1. Use sempre que possível substituir a parte interna de um laço excessivamente grande por chamadas de funções ou subrotinas.

§2. Na impossibilidade de introduzir chamadas de funções ou subrotinas utilize sempre *labels* para identificar laços grandes de forma a permitir busca pelo fim do mesmo e permitir verificar facilmente as partes de código que são internas aos laços.

R.CO.select_case - Sempre que possível prefira usar a construção “*select case*” ao invés de fluxos “*if/then/elseif/end if*”.

R.CO.where - Sempre que possível substitua laços de teste “*if/then/else*” pelo uso da instrução “*where*”.

```
where(chem_species_concentration(:, :, iSpc, iSrc) < 0.) &  
chem_species_concentration(:, :, iSpc, iSrc) = 0.
```

R.CO.passagem_parametro - A passagem de parâmetros nas chamadas de funções e subrotinas deverá ser referenciada pelo nome da variável interna na função ou subrotina. Usar a chamada dessa forma permite o uso da diretiva “*optional*” em qualquer ponto da chamada e é muito útil como documentação do fluxo do código. (veja o exemplo abaixo para a função *CalorSensivel*).

```
module modTst  
  implicit none  
  private  
  
  public :: calorSensivel  
  
contains  
  
  ! -----  
  function CalorSensivel(massa, calor_especifico, delta_temp) result (cs)
```

```

implicit none

real, intent(in) :: massa
!! massa da substância [g]
real, optional, intent(in) :: calor_especifico
!! calor específico da substancia [cal/gC]
real, intent(in) :: delta_temp
!! Temperatura de elevacao [C]

real :: cs
!! calor sensivel calculado

real :: local_calor_especifico

! Se não fornecido o valor do calor especifico usar o da agua
if(present(calor_especifico)) then
    local_calor_especifico = calor_especifico
else
    local_calor_especifico = 1.0
endif

cs = massa*local_calor_especifico*delta_temp

end function CalorSensivel

end module modTst

program teste
    use modTst, only: CalorSensivel
    implicit none

    write(*,*) CalorSensivel(massa = 4000.0, delta_temp = 100.0)
    write (*,*) raiz

end program teste

```

R.CO.retorno_função - O retorno da função sempre deve ser indicado pela diretiva “*result*”, pois permite trabalhar com retornos mais complexos como *arrays*, etc.

```

module modTst
    implicit none
    private

    public ::SolveGrau2, calorSensivel

```

```

contains
! -----
function SolveGrau2(a_coef, b_coef, c_coef) result(raiz)
  implicit none

  real, intent(in) :: a_coef
  !! Coef. a da equacao de seg. grau
  real, intent(in) :: b_coef
  !! Coef. b da equacao de seg. grau
  real, intent(in) :: c_coef
  !! Coef. c da equacao de seg. grau

  real :: raiz(2)
  !! raizes da equacao de segundo grau

  real :: delta
  !!Delta da equação

  delta = b_coef * b_coef - 4 * a_coef * c_coef
  raiz(1) = (-b_coef + sqrt(delta)) / 2 * a_coef
  raiz(2) = (-b_coef - sqrt(delta)) / 2 * a_coef

end function SolveGrau2

! -----

end module modTst

```

R.CO.tipos_derivados_parametros - Sempre que possível use tipos derivados (estruturas) para agrupar variáveis, vetores ou *arrays* e facilitar a codificação de chamadas de subrotinas e funções. Esta prática permite alterar os tipos derivados sem a necessidade de refatoração do código que invoca a rotina.

```

module modEnergy
...
  type t_En
    real, pointer :: calor_sensivel(:,:,:)
    real, pointer :: calor_latente(:,:,:)
    integer :: x_dimension
    integer :: y_dimension
    integer :: z_dimension
  end type t_en
  type(t_en) :: atmos_data

```

```

...
end module modEnergy
...
call calcHeat(atmos_data)

...
subroutine calcHeat(atmos_data)
...
  use modEnergy, only: &
    t_en

  type(t_en) :: atmos_data

...

```

R.CO.controle_fluxo_grandes - Estruturas de controle de fluxo if/elseif/else/endif muito grandes devem ser evitadas.

R.CO.argumentos_inválidos_parametros - Sempre que possível, em funções e subrotinas, verifique se há valores de argumentos inválidos. Atente que isso pode causar problemas com desempenho. Sempre leve em conta se podem haver erros catastróficos relacionados a argumentos inválidos e se esses erros são de difícil detecção.

R.CO.tratamento_erro - Tratamento de Condições de erro: Quando ocorre uma condição de erro dentro de uma função/procedimento, deve ser impressa uma mensagem descrevendo o que deu errado. O nome da rotina e do arquivo (*p_procedure_name* e *p_source_name*) em que ocorreu o erro deve ser incluído.

R.CO.goto_continue - Recomenda-se fortemente e sempre que possível que não se use o comando “goto” ou “continue” substituindo-os por estruturas que usem testes/desvios condicionais e os comandos “cycle” e “exit”. Esta regra melhora a legibilidade bem como contribui para a manutenibilidade do código, uma vez que *goto* e *continue* torna mais difícil o entendimento do fluxo de controle do programa.

R.CO.procedure_expert - Os procedimentos devem ser logicamente planos (devem se concentrar em uma funcionalidade específica, não em várias).

R.CO.funções_ponteiro - As funções não devem ter como resultados retornáveis um ponteiro.

R.CO.funções_nomes - Os nomes de funções intrínsecas (por exemplo, “Sum”) não devem ser usados para funções definidas pelo usuário.

R.CO.procedure_funções_retorno - Os procedimentos que retornam um único valor devem ser funções; observe que valores únicos também podem ser tipos derivados definidos pelo usuário (o que pode retornar um pacote de valores encapsulados - “tipos”).

5.3. Documentação (DO)

R.DO.ford - Todas as variáveis comunicadas nas chamadas de procedures, todas as constantes e todos os parâmetros (*parameters*) devem receber uma documentação pelo padrão de documentação FORD (2024) - FORtran Documentator , que é adotado pelo modelo. Essa documentação está descrita logo abaixo da declaração da variável com os caracteres “!” antes da descrição. Essa documentação deve ser o mais abrangente possível e, caso exista, deve conter também a unidade que é referência sobre cada variável.

R.DO.markdown - Todos os documentos relativos ao *software* (manuais, instruções de instalação, etc) devem ser editados em linguagem Markdown (2024) para facilitar sua portabilidade para outros sistemas operacionais e máquinas. Além disso, o formato usa apenas ASCII e é aceito nos controles de versionamento GIT.

R.DO.cabeçalho_módulos - Os cabeçalhos de módulos devem seguir o padrão FORD e devem ter o seguinte formato:

```
! -----  
module nome_do_módulo  
  !! ## Breve_descrição_do_módulo  
  !!  
  !!   
  !! ## MONAN  
  !!  
  !! Author: nome_do_autor(es)  
  !!  
  !! E-mail: <mailto:email_do_autor(es)>  
  !!  
  !! Date: data_de_criação  
  !!
```

```

!! #####Version: numero_da_versão_em_git_flow
!!
!! -
!! **Full description**:
!!
!! Descrição_completa_da funcionalidade_do_módulo (multilinha)
!!
!! ** History**:
!!
!! - Itenizado_as_alterações_ao_longo_do_tempo (genérica)
!!-
!! ** Licence **:
!!
!! 
!!
!! This program is free software: you can redistribute it and/or modify
!! it under the terms of the GNU General Public License as published by
!! the Free Software Foundation, either version 3 of the License, or
!! (at your option) any later version.
!!
!! This program is distributed in the hope that it will be useful, but
!! ** WITHOUT ANY WARRANTY **; without even the implied warranty of
!! **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**. See the, GNU
!! GNU General Public License for more details.
!!
!! You should have received a copy of the GNU General Public License
!! along with this program. If not, see [GNU Public
License](https://www.gnu.org/licenses/gpl-3.0.html).
!!

```

R.DO.cabeçalho_funções - Os cabeçalhos de funções devem seguir o padrão FORD e devem ter o seguinte formato:

```

! -----
function nome_da_função(variável_de_entrada(s)) result(variável_de_retorno)
  !! # Breve_descrição_da_função
  !!
  !! Author: nome_do_autor(es)
  !!
  !!E-mail: <mailto:email_do_autor(es)>
  !!
  !!Date: data_de_criação
  !!
  !!#####Version: numero_da_versão_em_git_flow
  !!
  !! -
  !!**Full description**:
  !!
  !!Descrição_completa_da funcionalidade_da_função (multilinha)
  !!
  !!** History**:
  !!
  !!- Itenizado_as_alterações_ao_longo_do_tempo (genérica)
  !!-

```

```
!!
!!
```

R.DO.cabeçalho_subrotinas - Os cabeçalhos de subrotinas devem seguir o padrão FORD e devem ter o seguinte formato:

```
! -----
subroutine nome_da_subrotina()
  !!## Breve descrição da subrotina
  !!
  !!Author: nome_do_autor(es)
  !!
  !!E-mail: <mailto:email_do_autor(es)>
  !!
  !!Date: data_de_criação
  !!
  !!#####Version: numero_da_versão_em_git_flow
  !!
  !!-
  !!**Full description**
  !!
  !!Descrição completa da funcionalidade da subrotina (multilinea)
  !!
  !!** History**
  !!
  !!- Itenizado as alterações ao longo do tempo (genérica)
  !!-
  !!
  !!
```

R.DO.comentários - Comente internamente o código de forma a se entender o que significa e a função específica de cada bloco. Se necessário documente até mesmo o que cada linha significa no contexto do *procedure*. Lembre-se que o código irá passar por manutenção constante e os comentários servirão como referência para outros programadores entenderem como ele foi construído.

R.DO.comentários_óbvios - Não use comentários óbvios como por exemplo “!! fazendo a variável *area* igual a $2 * c_pi * raio^2$ ” para uma linha de código `area = 2*c_pi*raio**2`. Prefira o comentário que explique a razão de um comando ou bloco de código como por exemplo “!! Determinando a área da circunferência a ser aplicada no cálculo da função *gamma* da seção transversal da radiação”.

R.DO.comentários_bloco - Quando um bloco de código resolve parte de uma equação ou sistema de equações use comentários que apontam ou expliquem a equação ou suas partes que serão resolvidas pelo bloco. Preferencialmente faça a devida referência da mesma com o cabeçalho do módulo (ou *procedure*) que deverá conter explicitamente a equação escrita usando a linguagem *markdown* - Esta linguagem é reconhecida pelo documentador Ford. Veja como fica o exemplo de SolveGrau2, mostrado em item acima, usando essa recomendação:

```
! -----
module modTeste
  !! ## Módulo criado como exemplo de teste
  !!
  !! 
  !! ## MONAN
  !!
  !! Author: Rodrigues, L.F. [LFR]
  !!
  !! E-mail: <mailto:luiz.rodrigues@inpe.br>
  !!
  !! Date: 14Outubro2022 08:33
  !!
  !! #####Version: 0.1.0
  !!
  ! -
  !! **Full description**:
  !!
  !! Módulo criado como exemplo de teste. Tem apenas o intuito de mostrar as
  !! boas práticas de programação que facilitam a manutenibilidade do código.
  !!
  !! ** History**:
  !!
  !!-
  !! ** Licence **:
  !!
  !! 
  !!
  !! This program is free software: you can redistribute it and/or modify
  !! it under the terms of the GNU General Public License as published by
  !! the Free Software Foundation, either version 3 of the License, or
  !! (at your option) any later version.
  !!
  !! This program is distributed in the hope that it will be useful, but
  !! ** WITHOUT ANY WARRANTY **; without even the implied warranty of
  !! **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**. See the, GNU
  !! GNU General Public License for more details.
  !!
```

```

use modConstants
implicit none
character(len=*), parameter :: p_source_name = 'exemplo.F90'
!! Nome do arquivo fonte
character(len=*), parameter :: p_module_name = 'modTeste'
!! Nome do módulo

private
public :: SolveGrau2

contains

function SolveGrau2(coeficiente) result(raiz)
!! ## Determina as raízes de uma função quadrática
!!
!! Author: Rodrigues, L.F. [LFR]
!!
!! E-mail: <mailto:luiz.rodrigues@inpe.br>
!!
!! Date: 14Outubro2022 08:34
!!
!! #####Version: 0.1.0
!!
!! -
!! **Full description**:
!!
!! Determina as raízes de uma função quadrática
!! A solução por bháskara é
!!  $x = \frac{-b \pm \sqrt{\delta}}{2a}$  [*1]
!! Onde
!!  $\delta = b^2 - 4ac$  [*2]
!!
!! Obs: se delta é negativo essa função não retorna valores das raízes
!! A função é abortada sem o cálculo de raízes imaginárias.
!!
!!
!! ** History **:
!!
!! -
!! -
!! 
!!

implicit none
!Parameters:
character(len=*), parameter :: p_procedure_name = 'SolveGrau2'
!! Nome da função
integer, parameter :: p_a=1,p_b=2,p_c=3
!! Índices dos coeficientes a,b e c da função
!! Apenas para teste

!Variables (input):
real, intent(in) :: coeficiente
!! Coeficientes de entrada [/a,b,c/] fornecidos em um array de 3 elementos

!Local variables:

```

```

real :: raiz(2)
!! Raízes da equação - retornam em um array de 2 elementos
real :: delta
!! Delta da equação (2)
integer :: err
!! Apenas para armazenar o valor do erro

! Calculando o valor de delta conforme eq. [*2]
delta = coeficiente(p_b)*coeficiente(p_b)-4*coeficiente(p_a)*coeficiente(p_c)

if(delta < 0) err = StopExecution("Sem solução real!", "modTeste.F90",
"SolveGrau2")

! Calculando a raiz 1 segundo a fórmula eq. [*1]
raiz(1) = (-coeficiente(p_b)+sqrt(delta))/2*coeficiente(p_a)
! Calculando a raiz 2 segundo a fórmula eq. [*1]
raiz(2) = (-coeficiente(p_b)-sqrt(delta))/2*coeficiente(p_a)

end function SolveGrau2

end module modTeste

```

5.4. Entrada/Saída - I/O (ES)

R.ES.unit_automático - O controle de número de unidade (*"unit"*) para abertura de arquivo deve ser feito por função automática que deve retornar a unidade livre. Da mesma forma, a unidade deve ser liberada por uma função específica que fecha o arquivo e libera a unidade.

R.ES.check_existência_arquivos - Precedendo a abertura de arquivos uma função deve ser usada para verificar a existência de diretório (em caso de escrita e leitura) e a existência do arquivo (em caso de leitura) e informar erro - ou aviso - em caso de não existência dos mesmos. Essa função também deve receber os parâmetros *p_source_name* e *p_procedure_name* para que se identifique qual arquivo e rotina fez a tentativa de operação.

R.ES.instruções_es_parâmetros - As instruções de E/S em arquivos externos devem conter os parâmetros do especificador de status *err=*, *end=*, *iostat=*, conforme apropriado.

R.ES.write_print - Use instruções *write* em vez de *print* para saída na tela (standard output).

R.ES.netcdf - Sempre que possível os arquivos de entrada e saída devem estar em formato NetCDF (2024). Um documento deve ser criado para referenciar cada uma das variáveis de cada um desses arquivos ao seu respectivo significado, sua unidade e outras informações

necessárias. Quando os arquivos de entrada estiverem em formato distinto, esses devem ser convertidos para NetCDF.

5.5. Formatação de Código (FC)

R.FC.camelcase - Nomes de subrotinas devem usar o estilo camelCase que consiste nas palavras que compõem o nome da subrotina com letras minúsculas, com a primeira letra sempre minúscula e as outras palavras começando com maiúsculas .

O uso consistente de um padrão de escrita para um programa de construção coletiva contribui para a manutenibilidade do código bem como para uma maior legibilidade.

Exemplo:

```
subroutine countParticles()  
...  
subroutine checkBounds()
```

R.FC.pascalcase - Nomes de funções devem usar o estilo PascalCase que consiste nas palavras que compõem o nome da função com letras minúsculas, com a primeira letra sempre maiúscula e as outras palavras começando com maiúsculas . Exemplo:

```
function CountParticles() result(number_particles)  
...  
function CheckBounds() result(bounds_ok)
```

A adoção consistente de um padrão de escrita em programas de construção coletiva contribui para a manutenibilidade do código bem como para uma maior legibilidade.

R.FC.snake_case_namelist - Todas as variáveis que devem ser lidas de um namelist devem usar SNAKE_CASE com todas as letras em maiúsculo. Exemplo:

```
real :: TEMPK_TMP(kmax)  
!! Temperatura da camada [Kelvin] (Temporária)  
character(len=32) :: p_source_name  
! !Nome do arquivo fonte  
integer :: OPTIONS(5)
```

R.FC.comandos_linha - O número máximo de comandos por linha de código é 1. Não se deve usar ponto e vírgula (“;”) para separar mais de um comando por linha. Ao utilizar um comando por linha essa prática aumenta a legibilidade do código.

```
write(*,*) "Na mesma linha"; call flush(6)  
write(*,*) "Em Linhas separadas"  
call flush(6)
```

Observação: nos casos de código usados apenas para *debugs* e que serão retirados em momento posterior pode-se não aplicar a regra.

R.FC.linha_limite - Uma linha não pode ter mais de 132 caracteres de tamanho e deve se usar a continuação de linhas para os casos que excedem esse tamanho.

§1. Sempre que possível mantenha a linha com menos de 80 caracteres.

Ao definir esses limites de caracteres contribui para legibilidade e manutenibilidade do código.

R.FC.quebra_linha_fórmulas - Continuidade em quebra de linha: Em fórmulas matemáticas, colocar o caractere de continuação “&” logo antes de um operador de matemática. Colocar o operador alinhado com o símbolo de “=”. Pretende-se com essa regra obter uniformidade do código e contribuir para legibilidade

```
formula = (fibon(i,j) + c_pi * angulo) &  
+ delta
```

R.FC.quebra_linha_string - Continuidade em quebra de linha: Caso uma linha contenha um *string* que precise continuar na linha seguinte deve-se usar a concatenação de string como ferramenta de continuação. Fecha-se os delimitadores de *string* e coloca-se o “&”. Na linha seguinte, alinhado com o início do *string*, coloca-se o carácter de concatenação “//” e termina-se de escrever o *string*.

```
character(len = *), parameter :: nome_arquivo = 'diretorio/files/dados/input/' &  
// 'arquivoEntrada'
```

R.FC.quebra_linha_argumentos - Havendo necessidade de quebrar linhas de argumentos de funções/subrotinas, colocar o carácter de continuidade de linha “&” logo antes de uma vírgula de separação de dado e na linha seguinte começar com uma “,” alinhada com o primeiro parênteses logo após o nome do procedure.

```
subroutine executaAcao(param1, param2 &
                    ,param3, param4)
```

R.FC.início_procedure - O início de procedures (*function, subroutine, module, program*) deve receber uma linha em branco, seguido de uma linha com comentário (“!”) seguido de um espaço e de 60 caracteres (ou mais) “-”. Na linha seguinte segue a declaração do nome do procedure e seus parâmetros de entrada e saída e logo depois deve receber a documentação padrão de cabeçalho do Ford.

```
! -----
function SomaArrays(aVar, bVar, nVar) result(saida)
    ...
```

R.FC.espaços_branco_linha vazia - Não podem ser deixados espaços em branco após as linhas contendo código (ao final da linha).

R.FC.comentários_espaço_branco - Em linhas de comentário deixe um espaço em branco entre o “!” e o primeiro carácter válido.

R.FC.chamadas_parâmetros - Nas chamadas de subrotinas ou funções os parâmetros devem ser passados separados por vírgula seguido por um espaço em branco:

```
call doSomeCalc(x, y, 35.0)
```

R.FC.espaço_nome_funcao parenteses - Não deixar espaço entre o nome da função ou subrotina e a abertura do parênteses que o segue.

R.FC.espaço_atribuições - Nas atribuições de valores a variáveis e constantes, bem como no uso de operadores lógicos deve-se observar um espaço antes e depois do símbolo empregado:

```
real :: distance  
  
distance = 500
```

R.FC.tipos_prefixo - Todos os tipos (*type construct*) devem receber o prefixo “t_” precedendo seu nome. Exemplo:

```
type t_En  
  real, pointer :: calor_sensivel(:, :, :)  
  real, pointer :: calor_latente(:, :, :)  
  integer :: x_dimension  
  integer :: y_dimension  
  integer :: z_dimension  
end type t_en
```

R.FC.declaração_variáveis_linha - Sempre que possível evite declarar mais de uma variável local numa mesma linha. Faça a declaração em linhas separadas e com os comentários FORD.

R.FC.indentação_comentários - Use o mesmo recuo (indentação) para comentários como para o restante do código.

R.FC.operadores_lógicos - O uso de operadores <, >, <=, >=, ==, /= é encorajado (para facilitar a leitura) em vez de .lt., .gt., .le., .ge., .eq., .ne. Estes operadores devem ser utilizados para números. Para comparar valores lógicos, é obrigatório o uso de .eqv. e .neqv.

R.FC.alinhamento_vertical - Alinhar verticalmente: atributos, variáveis, comentários dentro da seção de declaração de variáveis. Isso dá clareza ao código.

R.FC.identificadores_simples - Para qualquer programa, subrotina, função, módulo, variáveis, etc deve-se evitar identificadores simples ou de apenas uma ou duas letras para variáveis como *T, P, U*. Usar nomes como *temp, press* e *u_wind* por exemplo. Isso torna a

manutenção mais simples quando são necessárias buscas por variáveis dentro de códigos. Use os caracteres necessários, contudo tente não exceder o tamanho de 15 caracteres.

R.FC.identificadores_variáveis - Para variáveis auxiliares e locais aos *procedures* utilize identificadores que ajudem a identificá-las numa busca no código. Se possível, documente no padrão Ford. Por exemplo: para contadores de laços use o sufixo *cnt* (*count*) para identificá-los: Exemplo: *i_cnt, j_cnt*, etc. Evite mesmo nesses casos os caracteres simples (*i, j, k*, etc).

R.FC.identificadores_procedures - Os *procedures* devem ter identificadores significativos que facilitem entender suas funcionalidades. Exemplo: ao invés de “function *TU()*” use function “*ToUpper()*”

R.FC.identificadores_módulos - O nome dos módulos deve começar com as letras *mod*. A continuação do nome usa o estilo PascalCase. Caso exista apenas um módulo em um arquivo, o nome do arquivo fonte deve ser o mesmo do módulo. Exemplo: *modRadiate.F90*. O restante do nome do módulo deve usar uma *string* que descreve da melhor forma sua função principal.

```
module modRadiate  
  
end module modRadiate
```

R.FC.linhas_em_branco - O uso de linhas em branco no código são recomendadas. Mas seu uso excessivo pode tornar o código com aspecto visual ruim e de difícil compreensão. Por isso recomenda-se:

a) Duas linhas em branco:

- Entre funções / subrotinas;

b) Uma linha em branco:

- Dentro das funções / subrotinas, entre o escopo de variáveis e os blocos de código;
- Entre blocos lógicos dentro das funções / subrotinas.
- Entre seções importantes do código;

- Entre definições de dados dentro de um módulo.

5.6. Modularidade e Reuso (MR)

R.MR.modconstants - Todas as constantes físicas e não físicas **comuns a todos os arquivos fontes** do projeto, devem estar definidas em um módulo comum com o nome `modConstants.F90`. Esse arquivo deve sempre ser referenciado no início dos módulos através de um comando *“use modConstants”*.

```
use modConstants
```

§1. O código fonte `modConstants.F90` deve estar disponível no diretório *src (sources)*.

Esta prática contribui para manutenção do código ao necessitar alterações das constantes, pois todas as constantes estarão associadas somente ao módulo “modConstants.F90”.

R.MR.constante_identificação_rotinas - Deve-se introduzir em cada *procedure* duas constantes que serão necessárias para manutenção de código e arquivos de *dump*. São elas: *p_source_name* e *p_procedure_name*. A constante *p_source_name* deve conter o nome do arquivo `.F90` em que ela está contida e a constante *p_procedure_name* deve conter o nome da *procedure (function, subroutine, module, etc)*.

```
integer function WriteAHello()  
  implicit none  
  
  character(len=*), parameter :: p_source_name = 'writeAHallo.F90'  
  character(len=*), parameter :: p_procedure_name = 'writeAHello'  
  
  write(*,*) 'Hello'  
  
end function WriteAHello
```

Essas constantes serão usadas em funções de manutenção de código e verificação de *bugs* sempre que necessárias.

R.MR.módulo_private - Para o caso dos módulos deve-se sempre considerar que todo o módulo é “*private*”, exceto as variáveis e *procedures* explicitamente declaradas como “*public*”. Esta regra permite maior segurança dos dados com o encapsulamento.

R.MR.modularidade - Todas as parametrizações (códigos específicos) devem estar “encapsuladas” em um mesmo módulo (“*module*”). Os módulos devem conter suas próprias variáveis de memória no escopo do módulo. As variáveis e *procedures* devem ser declaradas levando-se em consideração seu escopo no nível do programa todo. Devendo ser:

- Pública (*public*): Pode ser visível em todo o código;do/end do
- Privada (*private*): Somente visível dentro do módulo.

R.MR.módulos_inicialização - Os módulos devem conter suas próprias funções de inicialização, se necessárias, e especialmente aquelas para zerar variáveis. Considerar sempre que a criação de uma variável não a inicia para valor algum. Essa função deve ter o nome começando com as letras *init*. Exemplo: *initModRadiate* e retornar o valor “0” quando a inicialização obtiver sucesso e outro valor indicando algum erro.

R.MR.módulos_use - Os módulos não podem fazer uso (*use*) de variáveis de outros módulos externos a ele. Somente as referências aos módulos globais é permitido (como o de constantes, *modConstants*). Para o ajuste de valores em uma variável em outro módulo deve ser usada uma função “*set*” criada no módulo. Para obtenção de valores de variáveis de outro módulos deve ser usada uma função “*get*” criada no módulo.

§1. Os uses permitidos devem ser usados sempre com as cláusulas “*only*”

R.MR.modconstants_errores - Todos os erros que retornam de funções devem ser listados dentro do módulo *modConstants.F90* e devem receber seus valores indicados, bem como sua descrição deve estar definida. Os erros são parâmetros (*parameters*) e devem começar com “*e_*”.

§1. Todos os erros devem ser listados em um *array* de *strings* dentro do arquivo “*modConstants.F90*” para que possam ser identificados.

```

!Errors:
integer, parameter :: e_error = 1
!Errors numbers
integer, parameter :: e_no_error = 0
!! No Error, It's fine
integer, parameter :: e_overflow = 1
!! Overflow Error
integer, parameter :: e_nan = 2
!! Not a Number Error
integer, parameter :: e_out_of_bounds = 3
!! Out of bounds error
integer, parameter :: e_invalid_string = 4
!! String not permitted error
integer, parameter :: e_invalid_temperature = 5
!! Temperature invalid (bellow zero)
character(len=32), dimension(6), parameter :: errors_description = (/ &
    "No Error, It's fine" & !0
    , "Overflow Error" & !1
    , "Not a Number Error" & !2
    , "Out of bounds error" & !3
    , "String not permitted error" & !4
    , "Invalid Temperature error" /) !5

```

R.MR.módulos_drivers - Módulos devem possuir interfaces nos *drivers* ou usar um sistema de acoplamento (acopladores). Os *drivers* servem para fazer conversões numéricas, de tipos de variáveis e adaptação entre os *procedures* que chamam o módulo e o próprio módulo chamado. O *driver* deve conter o nome composto do nome do módulo seguido da palavra *Driver*. Por exemplo, se um *driver* for criado para o módulo *modRadiate* este deverá se chamar *modRadiateDriver*. Os *drivers* podem conter “uses” de outros módulos.

R.MR.drivers_use - Todos os “use”s dos *drivers* (se existirem) devem conter a cláusula “only” e conter a lista de variáveis, tipos derivados e *procedures* a que ele referencia.

R.MR.drivers_parâmetros - Os *drivers* criados devem chamar os *procedures* do módulo sempre por parâmetros e nunca por memória (“use”, etc). Os módulos não podem fazer uso de variáveis ou *procedures* com “use”.

R.MR.módulos_erro - Deve ser criado um módulo com rotinas específicas para a comunicação de erros ou outras mensagens necessárias ao MONAN. Esse módulo pode ser usado (*use*) por outros módulos no MONAN.

R.MR.módulos_genéricos - Podem ser criados módulos com rotinas específicas não abrangidas pela linguagem Fortran para manipulação de *strings*, funções matemáticas, manipulação e controle de arquivos que poderão ser usados por outros módulos do MONAN.

§1. Funções identificadas dentro de códigos que tenham objetivos genéricos e que possam ser aproveitados por outros módulos devem ser retiradas destes códigos e movidas para os módulos com as rotinas especificadas acima.

R.MR.módulos_include - Nenhum arquivo de include deve ser usado. Em vez disso, use módulos, com instruções “*use*” em programas de chamada.

R.MR.módulos_variáveis_encapsulamento - As variáveis “*public*” do módulo (variáveis globais) devem ser usadas com cuidado e principalmente para dados estáticos ou com variação infrequente. Não abuse do número de variáveis globais. Variáveis devem estar encapsuladas ao módulo. Prefira usar funções de “*set*” e “*get*” para acessar variáveis internas ao módulo. Veja regra “*módulos_variáveis_declaração*”.

R.MR.reuso - O *software* deve sempre utilizar funções e *procedures* já existentes.

§1. Sempre que possível essas funções/*procedures* devem ser unidades atômicas reaproveitáveis e testáveis.

§2. Pode ser interessante que uma rotina seja transformada em função mesmo quando há mais de um argumento de retorno. É importante que a função retorne um valor booleano para identificar o sucesso ou falha, permitindo que a função possa ser testada por um código de teste ou pelo código que a invoca.

R.MR.módulos_variáveis_declaração - preferencialmente todas variáveis (*arrays*, tipos, etc) devem ser declaradas privadas (*private*) nos módulos. Para acessá-las ou atribuir valores

para as variáveis dos módulos devem ser usadas funções de “*set*”, para ajustar valores de variáveis internas ao módulo e, “*get*” para as mesmas.

R.MR.módulos_estruturas - Estruturas (tipos derivados) devem ser definidas dentro de seu próprio módulo. Os procedimentos para manipular essas estruturas também devem ser definidos dentro do módulo. Isto forma uma entidade semelhante a um objeto (Orientação a Objeto).

Referências bibliográficas

ATOM. Disponível em: <<https://github.blog/news-insights/product-news/sunsetting-atom/>>. Acesso em: 2 out. 2024.

BRING, e-commerce. Qualidade do código e sua importância para um desenvolvimento bem sucedido. Disponível em: <<https://bring.com.br/blog/en/qualidade-do-codigo-e-sua-importancia-para-um-desenvolvimento-bem-sucedido/>>. Acesso em: 2 out. 2024.

CARULLO, Giuliana. Implementing Effective Code Reviews: How to Build and Maintain Clean Code. Apress. ISBN 978-1-4842-6161-3e-ISBN 978-1-4842-6162-0. Ano 2020. <https://doi.org/10.1007/978-1-4842-6162-0>

FORD. Automatically generates FORtran Documentation from comments within the code - GitHub. Disponível em: <<https://github.com/Fortran-FOSS-Programmers/ford>>. Acesso em: 2 out. 2024.

FORTRAN. Fortran 90/95 Coding Conventions. Disponível em: <<https://alm.engr.colostate.edu/cb/wiki/16983>>. Acesso em: 2 out. 2024.

GCC. GCC Coding Conventions. Last modified 2024-09-17. Disponível em: <<https://www.gnu.org/software/gcc/codingconventions.html#Documentation>>. Acesso em: 2 out. 2024.

GNU. GNU Coding Standards. Retrieved 2020-11-29. Disponível em: <<https://www.gnu.org/prep/standards/standards.html#Formatting>>. Acesso em: 2 out. 2024.

ISO/IEC, INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 1539-1:1997 Information technology - Programming languages - Fortran - Part 1: Base language. Disponível em: <<https://www.iso.org/standard/26933.html>>. Acesso em: 2 out. 2024.

ISO/IEC, INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 9126: Software engineering - Product quality. 2001. Geneva: ISO, 2001.

ISO/IEC, INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 1539-1:2010 Information technology - Programming languages - Fortran Part 1: Base language Disponível em <<https://www.iso.org/standard/50459.html>>. Acesso em: 2 out. 2024.

ISO/IEC, INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. 2011. Geneva: ISO, 2011.

MARKDOWN. Getting Started: An overview of Markdown, how it works, and what you can do with it. Disponível em: <<https://www.markdownguide.org/getting-started/>>. Acesso em: 2 out. 2024.

MPAS. MPAS Developers Guide. Disponível em:
<<https://mpas-dev.github.io/files/documents/MPAS-DevelopersGuide.pdf>>. Acesso em: 2 out. 2024.

NETCDF. Network Common Data Form (NetCDF) - Unidata. Disponível em:
<<https://www.unidata.ucar.edu/software/netcdf/>>. Acesso em: 2 out. 2024.

PHOTRAN. Eclipse Photran Fortran Development Tools. Disponível em:
<<https://www.eclipse.org/photran/>>. Acesso em: 2 out. 2024.

SUBLIME. Sublime Text. Disponível em: <<https://www.sublimetext.com/3>>. Acesso em: 2 out. 2024.

VSCODE. Visual Studio Code. Disponível em: <<https://code.visualstudio.com/>>. Acesso em: 2 out. 2024.

Referências consultadas

FLYNN, Kathleen M. *et al.* A FORTRAN coding convention for use in the u.s. geological survey, water resources division. u.s. geological survey Open-File Report 94-501. Ano 1994. Disponível em: <<http://water.usgs.gov/software/code/general/sysdoc/doc/coding.pdf>>. Acesso em: 2 out. 2024.

FORTTRAN. FORTRAN Coding Standards. Disponível em: <<http://dbwww.essc.psu.edu/lasdoc/programmer/4fortran.html>>. Acesso em: 2 out. 2024.

KERNEL. Linux kernel coding style - The Linux Kernel documentation. Retrieved 2017-10-12. Disponível em: <<https://www.kernel.org/doc/html/v4.10/process/coding-style.html>>. Acesso em: 2 out. 2024.

McCONNELL, Steve. Code Complete: A practical handbook of software construction. Redmond, WA: Microsoft Press. pp. 746–747. ISBN 0-7356-1967-0. Ano 2004. Disponível em: <<https://archive.org/details/codecomplete0000mcco/page/746>>. Acesso em: 2 out. 2024.

WRF. WRF Coding Conventions - Draft. Disponível em: <http://box.mmm.ucar.edu/wrf/WG2/WRF_conventions.html>. Acesso em: 2 out. 2024.