Modelo de Desenvolvimento Computacional para Modelos Numéricos do Instituto Nacional de Pesquisas Espaciais.

Rev 1.0

Barbara Yamada, Daniel Lamosa, Denis Eiras, Eduardo Khamis, João Gerd, Luiz Flávio

Setembro - 2018

1. Introdução

Modelos numéricos, como todo tipo de software, possuem ciclos de vida bem definidos. O **ciclo de vida** de um software é a estrutura contendo processos, atividades e tarefas envolvidas no desenvolvimento, operação e manutenção de um produto de **software**, abrangendo a **vida** do sistema, desde a definição de seus requisitos até o término de seu uso [1].

Sabe-se que a escolha do modelo de ciclo de vida de um software é fundamental para mantê-lo saudável, funcional e, dentro das possibilidades, o mais isento de erros (*bugs*) que possam comprometer o seu funcionamento. Para tanto, se faz necessário adotar um ciclo, adequado à equipe disponível, para seu desenvolvimento e manutenção. O que se busca é garantir que sejam atendidos não apenas os requisitos funcionais (RF) do software mas também os requisitos não funcionais (RNF), sendo a qualidade do software definida como soma de RF e RNF.

Na literatura existem inúmeros modelos para definir o ciclo de vida de um software, dos quais podem ser citados alguns:

- Cascata
- Modelo em V
- Incremental
- Evolutivo
- RAD
- Prototipagem
- Espiral
- Modelo de Ciclo de Vida Associado ao RUP

Alguns desses modelos são antigos, por exemplo o Cascata é de 1970, e outros usam características mais adaptadas a novos padrões de desenvolvimento de software. Uma análise da forma como o CGCPT tem desenvolvido seus modelos numéricos mostra que existe uma interação muito profunda entre o cliente (pesquisador) e o desenvolvedor. Na maioria das vezes os dois se fundem em uma só pessoa fazendo com que o próprio pesquisador desenvolva partes do software dentro do ciclo de vida do mesmo. Em geral pessoas com formação em computação atuam após a detecção de erros ou quando verifica-se que a performance não é adequada para a arquitetura da máquina onde o modelo numérico será executado.

Ao trabalhar com os modelos numéricos nota-se que os mesmos possuem pouca, ou nenhuma, documentação e utilizam, muitas vezes, técnicas inadequadas de programação. Isto é resultado de fatores como:

- Urgência de entrega de uma nova funcionalidade

- Falta de conhecimento de técnicas de programação moderna
- Desconhecimento de engenharia de software
- Ausência de controle e acompanhamento de desenvolvimento
- Falta de revisão do código por terceiros

O que se espera com a adoção de um ciclo de vida é que no mínimo o software ou modelo numérico cumpra, o mais próximo possível, o determinado pela norma NBR ISO/IEC 9126-1, ISO/IEC 25010 que tratam da qualidade de software sem comprometer de maneira impactante os tempos e prazos de desenvolvimento.

No texto que segue consideramos cliente e pesquisador ou operação como sendo uma mesma pessoa e tratamos software como modelos (numéricos) e suas partes (pré-processamento, pós-processamento). Também mostramos o ambiente de produção que chamamos como operação.

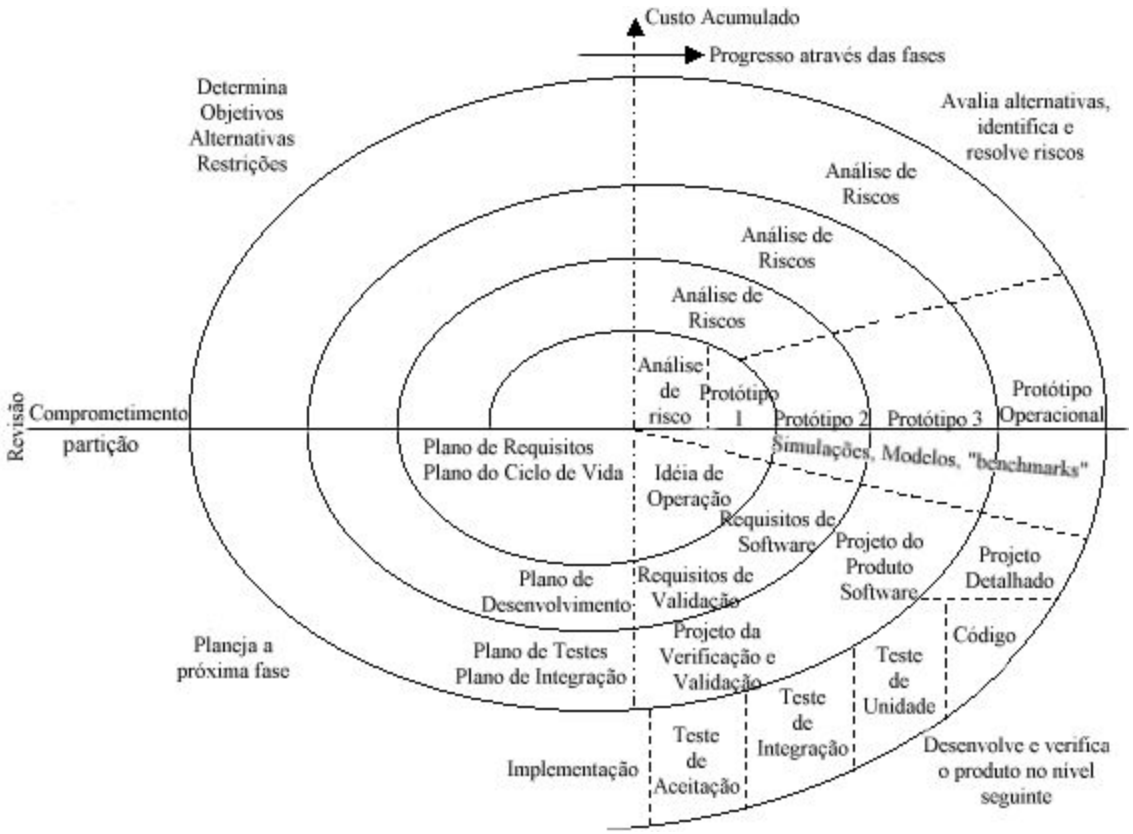
2. Ciclo de vida de software adotado: Evolutivo Espiral

O modelo evolutivo considera que o pesquisador não expõe todos os requisitos necessários para o funcionamento do modelo e estes são parte de um ciclo constante de desenvolvimento. Os desenvolvedores entregam um modelo para a operação e estes perfazem dois períodos sendo o primeiro de avaliação e um segundo de operação contínua. Durante qualquer um desses períodos o modelo pode apresentar necessidades de:

a) Solução de erros operacionais;
b) Introdução de uma funcionalidade necessária para a correta operação;
c) Introdução de novas funcionalidades para desenvolvimento pelo pesquisador.

A operação ou pesquisador dá um *feedback* sobre tais necessidades e um ciclo de nova análise, projeto e desenvolvimento são realizados, e uma nova versão do modelo é entregue ao pesquisador/operação.

O modelo Evolutivo é considerado adequado quando a interação com o pesquisador/operação é constante, caso do CGCPT e essa interação evita interpretações inadequadas e gera um código mais próximo das necessidades do cliente. O ciclo de vida é representado na figura abaixo:



Tratamos o modelo de ciclo de vida como **evolutivo + Espiral** pois o ciclo evolutivo por definição não trata adequadamente a documentação do software e não requer a fase de análise pré-operação. Contudo entende-se que a documentação é fator primordial para qualidade de software e o CGCPT historicamente adota um sistema de testes antes que o modelo entre em operação 'oficial'. Essa é a fase adotada pela Implantação Operacional (IO).

Segundo documentação do site DevMedia[2], esse modelo é "*guiado por risco, suporta sistemas complexos e/ou de grande porte, onde falhas não são toleráveis. Para isso, a cada iteração há uma atividade dedicada à análise de riscos e apoiada através de geração de protótipos, não necessariamente operacionais para que haja um envolvimento constante do cliente nas decisões*". Define ainda que o modelo de ciclo de vida possui 4 setores que são i) viabilidade do projeto, ii) Definição de requisitos do sistema, iii) Projeto do sistema e iv) Desenvolvimento e teste de unidade para somente depois o modelo se tornar operacional. Continua o site explicando cada setor:

1. *Na Definição de Objetivos, desempenhos, funcionalidade, entre outros objetivos, são levantados. Visando alcançar esses objetivos são listadas alternativas e restrições, e cria-se um plano gerencial detalhado.*
2. *Na Análise de Riscos, as alternativas, restrições e riscos anteriormente levantados são avaliados. Neste setor (porém não apenas neste) protótipos são utilizados para ajudar na análise de riscos.*

3. *No Desenvolvimento e Validação um modelo apropriado para o desenvolvimento do sistema é escolhido, de acordo com o risco analisado no setor anterior (cascata, interativo,...).*
4. *No Planejamento da Próxima fase ocorre a revisão do projeto e a decisão de partir para a próxima fase.*

Cabe citar em uma pequena análise que o uso de protótipos é especialmente adequado ao sistema já adotado no CGCPT visto que é comum que módulos inteiros sejam importados de outros centros mundiais de meteorologia não estando necessariamente adequados a serem 'plugados' aos modelos em desenvolvimento no CGCPT cabendo portanto testes dos protótipos de forma a se analisar os riscos de implementação dos mesmos. Vale ainda ressaltar que o modelo de desenvolvimento a ser utilizado nasce especificamente da análise dos riscos levantados. Dessa maneira é possível escolher uma metodologia de desenvolvimento em engenharia de software adequado, o que em geral deve ser um método ágil, como Extreme Programming (XP) ou Scrum.

Nos nós de cada volta da espiral são lançadas versões de desenvolvimento (Beta) até que uma versão operacional (*release tag*) seja disponibilizada. Para tanto verifica-se a fundamental importância de um sistema de controles de versões e de um sistema de controle de documentação e projeto.

3. Metodologia de Desenvolvimento Ágil

O CGCPT, sendo parte de uma instituição pública de pesquisa, tem dificuldade de agregar mão de obra. Especialmente com a cada vez mais escassa possibilidade de concursos públicos. Devido a esse fator e outros é preciso adotar sempre que possível uma metodologia de desenvolvimento com foco em agilidade de equipes e qualidade de projetos, apoiada em valores como simplicidade, comunicação, *feedback* e coragem. Esses pontos são garantidos pelos métodos de desenvolvimento ágeis, especialmente o XP.

O desenvolvimento de modelos numéricos no CGCPT adota um modelo XP modificado como base fundamental na metodologia de desenvolvimento de software. Alguns pontos que diferem do XP tradicional são destacados no texto mais abaixo. Contudo alguns valores relativos a metodologia serão seguidos, sendo eles:

- A **simplicidade é necessária** desde a forma como se levanta requisitos até a codificação e os testes da solução desenvolvida;
- A **comunicação é obrigatória** para que não haja lacunas em processos e problemas entre equipe, cliente e fornecedor;
- O ***feedback* é a prática** fundamentada em retornar informações entre os membros da equipe e também na relação com o cliente, desde responder e-mails, telefonemas e demais meios. Devido a isso, é um mecanismo para melhorar a prática de comunicação explanada acima;

- E a **coragem para saber dizer NÃO** quando necessário, ou então para dizer que o projeto vai demorar além do estimado, pois os novos requisitos precisam ser codificados ou o código já em funcionamento precisa ser refatorado.

Adotam-se todas as práticas do XP entretanto três delas são modificadas e estão especificadas abaixo:

- **Pair Programming:** apesar de apontarmos na direção de uma programação em duplas observa-se que no desenvolvimento dos modelos numéricos essa programação se dará entre um analista/programador e um pesquisador/Cliente da área. Não necessariamente entre dois programadores como requer a metodologia. Recomenda-se que os dois trabalhem em parceria, preferencialmente sentados lado-a-lado, contudo esse trabalho pode ser feito com mesas e cadeiras separadas desde que sejam trocados os necessários feedbacks de forma constante em reuniões rápidas.

- **Move People Around:** O rodízio de pessoas não é adequado no XP modificado. Nesse caso usaremos um sistema de verificação de código por **revisão aos pares** onde o código alterado será enviado para análise de outra dupla que deverá apontar falhas de codificação, respeito às normas de codificação e documentação adequada. O código deve ser devolvido à dupla original para as devidas correções e esse ciclo deve se repetir até que o código esteja respeitando as normas definidas (Coding Standards - Padronização do código)

- **Planning Game (Planejando o jogo):** devem haver reuniões constantes entre os clientes e os desenvolvedores para tratar do andamento dos projetos. O XP padrão pede rápidas (10-15 min) reuniões diárias de preferência em pé (**StandUp meetings**). Como pedimos um pair programming alternativo essas reuniões devem ser realizadas diariamente entre os membros do projeto do modelo e em um ciclo mensal com os demais pesquisadores envolvidos no desenvolvimento.

As demais regras de XP serão mantidas:

- **Small Releases (Pequenas versões):** Sempre que possível deverão ser adotadas pequenas modificações a cada release do código que deverão ser bem documentadas e estar notadamente comentadas no site de controle de projetos.

- **Acceptance Tests (Testes de Aceitação):** Devem ser definidos padrões de aceitação para cada release ou modificação. Esses padrões irão gerar os testes a serem aplicados intermediariamente ou ao final do desenvolvimento do

modelo. Por exemplo: casos específicos de fenômenos atmosféricos conhecidos cuja resposta dos modelos irá afetar.

- **Test First Design (Primeiro os testes):** Testes devem ser possíveis sempre e baseados nas regras de testes de aceitação. Preferencialmente os códigos devem ser modulares cujos módulos (pequenos) sejam passíveis de testes simples. Por exemplo: um sistema de matrizes esparsas de equações diferenciais parciais pode ser testado separadamente do caso específico do seu uso em um modelo numérico.

- **Continuous Integration (Integração Contínua):** Os módulos desenvolvidos devem ser testados e integrados a outros módulos sempre que possível de forma que possam ser constantemente reutilizados. A documentação é importante pois os mesmos serão colocados em 'prateleira' para serem usados por outros desenvolvedores sempre que possível. O uso contínuo e a depuração dos mesmos garante o aumento de qualidade e redução de bugs associados.

- **Simple Design (Simplicidade de Projeto):** o código obrigatoriamente deve seguir uma forma de codificação simples e clara, bem documentada a cada passo do desenvolvimento e seguir as normas de design padrão da equipe de desenvolvimento.

- **Refactoring (Refatoração - melhoria constante do código):** Durante o desenvolvimento de cada nova funcionalidade, correção de bugs ou atividade cada desenvolvedor deve obrigatoriamente e constantemente corrigir falhas de design nas partes em que ele trabalha, especialmente quanto a trechos que não foram notados na revisão dos pares garantindo um código limpo e de fácil compreensão. A criação de testes automatizados (ver item 4.4) garante que alterações como o refactoring, ou outras alterações possam comprometer a qualidade do código;

- **Collective Code Ownership (Propriedade coletiva - O código é de todos da equipe):** todos tem o direito de modificar o código sem que seja comunicado aos demais membros da equipe, pois ele é parte da equipe de desenvolvimento. Entretanto após a modificação a revisão por pares e os testes são obrigatórios.

- **Coding Standards (Padronização do código):** O código deverá seguir padrões rígidos de codificação. Todos devem zelar pelo uso do padrão definido tanto no tempo de desenvolvimento, durante a revisão por pares e na refatoração. As regras são definidas mais adiante nessa documentação.

- **40 Hour Week (Otimizando as jornadas de trabalho):** As horas de trabalho de cada colaborador do desenvolvimento do modelo não excederá de forma alguma

um total de 40 horas por semana. Inclui-se nesse tempo as reuniões de projeto, tempo para descontração durante o dia de trabalho e atividades extras ao projeto que devem ser ao máximo evitadas.

- **The Customer is Always Available (O cliente sempre disponível):** Especialmente no desenvolvimento dos softwares complexos que são os modelos usa-se a disponibilidade do cliente para dirimir dúvidas quanto a problemas identificados na codificação. Também isso pode ser feito diariamente pelas stand-ups meetings.

- **Metaphor (Uso de metáforas no projeto):** ter sempre em mente que os analistas e programadores não são necessariamente físicos, matemáticos ou meteorologistas. Dessa forma o uso de metáforas que 'traduzam' os trechos de código, todo um projeto ou funcionalidades é recomendado.

Um complemento às práticas do XP, muito comum em ambientes acadêmicos, é a inclusão do:

- **Journal Club:** reuniões periódicas de 1 hora de duração, de preferência semanais, no formato de *journal club* onde um artigo técnico ou acadêmico é selecionado para ser debatido, de preferência um texto curto que é previamente estudado. Também podem ser incluídos softwares, algoritmos, técnicas de paralelização de códigos e novas arquiteturas de hardware. A escolha do texto pode ser feita de forma rotativa pelos integrantes do grupo e colaboradores.

4. Controle de projetos de modelos, documentação e Gestão de versões

4.1 Gestão

Todos os projetos serão gerenciados pelo site de projetos do CGCPT, ***Redmine***, https://projetos.cptec.inpe.br/ que estará dividido por projeto/modelo específico. Cada um deles terá um líder e seus desenvolvedores. Os mesmos terão gerenciamento de versões usando o ***Subversion*** (SVN) já acoplado ao atual sistema de gestão de projetos.

Os projetos terão tarefas definidas para cada desenvolvedor com o tempo de execução determinado em uma reunião que deverá tratar da subdivisão do mesmo bem como estimar os tempos de duração de tarefas e sub-tarefas. Se possível deve-se registrar os riscos específicos para a realização das tarefas e potencial fonte de mitigação. Dizer não é necessário quando se percebe que uma tarefa pode não ser cumprida no prazo estipulado desde que baseada em razões plausíveis.

Obrigatoriamente devem-se definir as duplas de desenvolvimento (pesquisador/desenvolvedor ou desenvolvedor/desenvolvedor) já nas reuniões iniciais e os tempos devem ser acordados entre eles.

Deve haver um único responsável (líder) que deve verificar diariamente nas stand-ups meetings o andamento das tarefas e de todo o projeto, tratar de atrasos com os membros da equipe e manter o sistema atualizado.

4.2 Controle de Versões

Os projetos terão versões de trabalho no trunk, versões de verificação (release candidates) em um branch svn e as versões releases em tags.
A numeração adotada para as versões se divide em três números inteiros separados por pontos. O primeiro valor inteiro é destinado para modificações maiores que englobam grandes alterações de código. O segundo valor trata de novos desenvolvimentos nos modelos e o terceiro trata de correções menores ou correção de bugs. Ex:

UM 5.4.12

Obrigatoriamente os commits realizados no controle de versões deverá ser bem documentado indicando os motivos do mesmo. Deve-se evitar comentar o óbvio.

4.3 Documentação de código

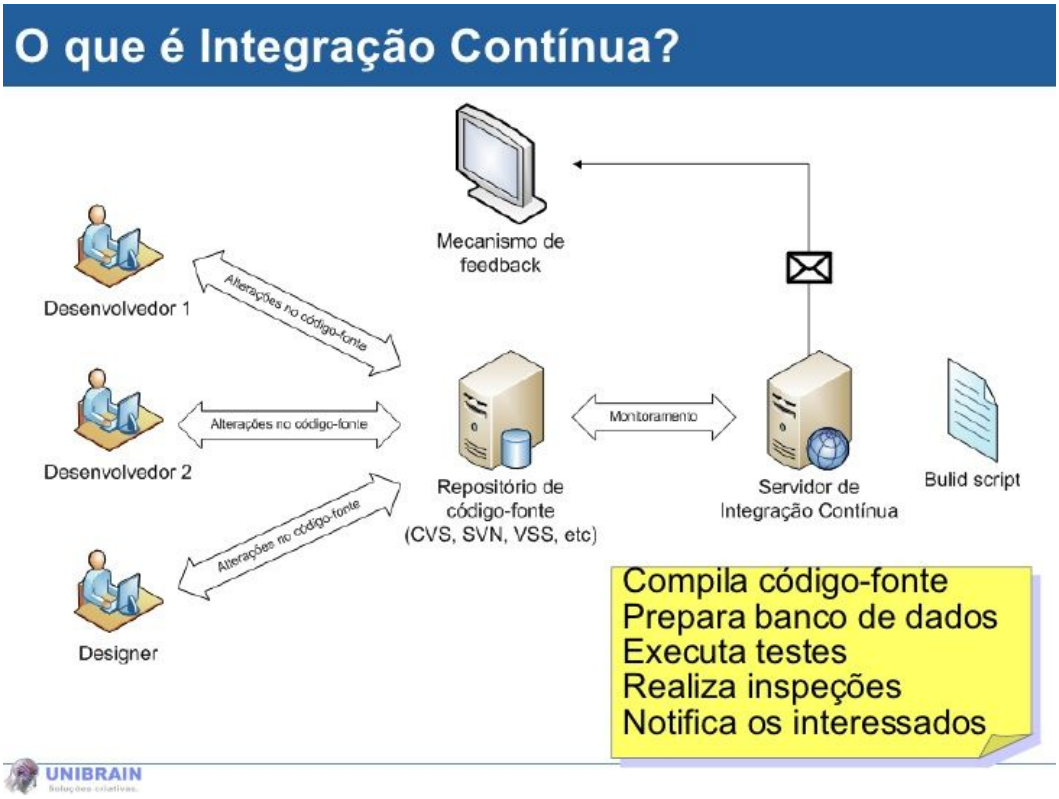
Para a documentação automática do código será usado o ***Ford***, Fortran documentator. https://politicalphysicist.github.io/ford-fortran-documentation.html. Esse sistema foi especialmente desenvolvido para documentar código em fortran e gera páginas web dinâmicas que permitem entender o funcionamento de funções, subrotinas, módulos e programas em fortran facilitando sobremaneira o entendimento e manutenção do código.

Esse já é um sistema existente e disponível para uso no CGCPT e as regras para a documentação estão definidas em uma área específica no site de gestão de projetos. A cada release estável (tag) uma documentação completa é gerada de forma automática e disposta em uma página HTML para consulta.

4.4 Integração Contínua e Sistema de Testes Automático

A Integração Contínua dentro da visão de gestão do produto refere-se à integração constante do código no repositório SVN, de forma que pequenas alterações são menos passíveis de falhas do que grandes alterações.

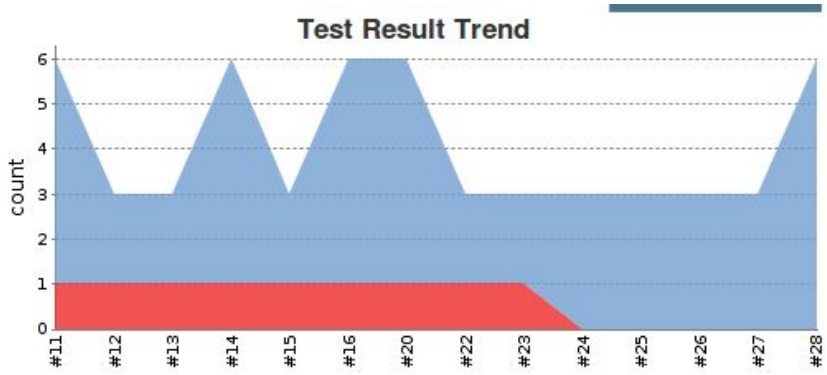
A automatização de testes será feita com o sistema ***Jenkins https://jenkins.io/***. (JENKINS, 2018). O jenkins é uma ferramenta que automatiza tarefas e um servidor de integração contínua bem estabelecida na comunidade de desenvolvimento que permite automatizar compilação de builds após integração do código, gerar versões automaticamente e armazenar os artefatos gerados das versões (executáveis, bibliotecas) para serem utilizados nos testes automatizados de software ou simplesmente ser disponibilizado para o usuário. Com ele é possível, por exemplo, executar scripts linux que rodam o modelo após um commit, gerar figuras que exibem o resultado da alteração integrada, tudo de forma automática, garantindo que o modelo executará normalmente e que os resultados são esperados, sem que seja necessário ter que executar todas essas tarefas manualmente, como pode ser visto na figura abaixo:



Todo o código ao qual se dê um commit no sistema de versões será automaticamente compilado e testado num set de testes mínimos, que devem ser pré-estabelecidos, que garantam sua estabilidade e qualidade. Os testes que falham após uma alteração no código, devem ser corrigidos, no caso de um cenário não previsto, ou o código integrado contém um bug previsto no teste, sendo necessária a correção do código nesse caso.

O Jenkins gera logs de execução de testes bem interessantes, mostrando os testes que executaram com sucesso e os que falharam (vermelhos), como pode

ser visto na figura abaixo. Também exibe o tempo de execução, commits que dispararam o build, dentre outros. O Eixo Y representa a quantidade de testes executados, onde a faixa vermelha representa os testes que falharam, e o eixo X representa o número do build gerado:



A implementação de testes automatizados, juntamente com o código a ser testado, devem cobrir determinados cenários previstos, garantindo que o código não irá falhar após alterações posteriores. Os testes automatizados também servem para alertar sobre um possível cenário não previsto, como parâmetros com valores indesejáveis que são passados para dentro do código coberto pelo teste, fazendo com que o teste falhe nesses casos e avisando o desenvolvedor via e-mail que o código deve tratar esse tipo de situação.

No Jenkins podem ser criadas quantas tarefas sejam necessárias, como pode ser visto na figura abaixo:

Também é possível encadear tarefas, tornando uma tarefa dependente da outra. Após um commit no SVN, é disparada uma primeira tarefa, denominada “trunk”, que realiza a compilação do código completo e gera o executável. Caso essa tarefa seja executada com sucesso, a tarefa “trunk test” é executada. Essa tarefa

consiste na execução de alguns testes, utilizando o executável gerado na tarefa anterior:

All Tests

Class	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
meteoChemInitialSuiteParallel	13 min	0		0		1		1	
meteoChemMakesfcSuiteParallel	7.5 sec	0		0		1		1	
meteoChemMakevfileSuiteParallel	11 sec	0		0		1		1	
meteoOnlyInitialSuiteParallel	1 min 56 sec	0		0		1		1	
meteoOnlyMakesfcSuiteParallel	5.3 sec	0		0		1		1	
meteoOnlyMakevfileSuiteParallel	8.2 sec	0		0		1		1	

5. Regras de codificação padrão (*Coding Standards*)

O padrão de codificação que será usado no desenvolvimento dos modelos numéricos do CGCPT estão definidos nos itens abaixo. Cada item recebe duas classificações:

Mandatória: Regras obrigatórias a serem seguidas

Recomendada: Sugestões de escrita fortemente recomendadas.

Para destacar os códigos do restante do texto foi adotado o critério de mudança de fonte dos trechos de interesse para *Courier New*. Exemplo:

```
program Teste
  implicit none
end program Teste
```

Recomendada: Os códigos em FORTRAN devem ser compatíveis com o padrão FORTRAN 90 ou superior. Códigos legados (padrão FORTRAN 77 e anteriores) devem ser migrados ou trocados por novas implementações.

Mandatória: As palavras reservadas da linguagem devem ser todas declaradas com letras minúsculas.

Padrão FORTRAN 77:

```
assign, backspace, block data, call, close, common, continue, data,
dimension, do, else, else if, end, endfile, endif, entry,
equivalence, external, format, function, goto, if, implicit,
inquire, intrinsic, open, parameter, pause, print, program, read,
return, rewind, rewrite, save, stop, subroutine, then, write.
```

Padrão FORTRAN 90:

```
allocatable, allocate, case, contains, cycle, deallocate,
elsewhere, exit, include, interface, intent, module, namelist,
nullify, only, operator, optional, pointer, private, procedure,
public, recursive, result, select, sequence, target, use, while,
where.
```

Padrão FORTRAN 95:

```
elemental, forall, pure.
```

Padrão FORTRAN 2003:

```
abstract, associate, asynchronous, bind, class, deferred, enum,
enumerator, extends, final, flush, generic, import,
non_overridable, nopass, pass, protected, value, volatile, wait.
```

Padrão FORTRAN 2008:

```
block, codimension, do concurrent, contiguous, critical, error
stop, submodule, sync all, sync images, sync memory, lock, unlock.
```

5.1. Declarações

Mandatória: Obrigatória a declaração de `implicit none` no início de programas, módulos ou na saída de uma *unit*. Ajuda ao programador na captura de erros como, por exemplo, saber se l (em destaque no exemplo a seguir) representa o número 1 ou uma variável:

Antes:

```
integer :: i
real, dimension(:) :: x(10)

do i = 2, 10
  x(i) = 2.0 * x(i-1)
end do
```

Depois:

```
integer :: i
Integer :: l
real, dimension(:) :: x(10)

do i = 2, 10
  x(i) = 2.0 * x(i-l)
end do
```

5.2. Nomes de Variáveis - Estilo Camelo (*Camel Case*):

Recomendada: Para qualquer programa, subrotina, função, módulo, etc deve-se evitar nomes simples ou de apenas uma letra para variáveis como T,P,U. Usar nomes como *temp*, *press* e *windU* por exemplo.

Mandatória: Nomes compostos com mais de uma palavra devem usar o estilo *camel case*, isto é, a primeira palavra em minúscula e as demais iniciando com maiúsculas. Exemplo:

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

5.3. Argumentos e Intents

Mandatória: O atributo `intent` deve ser usado para todos argumentos com exceção de ponteiros (não é definido pelo padrão). Deixa clara a intenção do argumento dentro da subrotina.

Antes:

```
subroutine test(a, b)
  real :: a
  real :: b
  a = b
end subroutine test
```

Depois:

```
subroutine test(a, b)
  real, intent( in) :: a
  real, intent(out) :: b
  a = b
end subroutine test
```

Para o código inicial, o compilador não gera nenhum erro em tempo de compilação, mas a introdução do intent no segundo trecho mostra qual seria a verdadeira intenção do programador, ou seja, a variável **a** sendo usada apenas como entrada na subrotina e a variável **b** tendo que ser carregada para que seu valor seja usado fora da subrotina. Dessa forma, o compilador pode gerar um erro em tempo de compilação (atribuição de a) e um aviso (b usado mas não foi inicializado).

5.4. Parameters

Mandatória : Constantes precisam ser definidas com o atributo `parameter`. Para declarações, utilize sempre uma constante por linha para facilitar a documentação e entendimento.

Mandatória :Iniciar com uma letra **c** seguida de um subscrito "**c_**".

Mandatória : Sempre coloque um comentário explicando a variável.

Mandatória : Definir como constantes, ao invés de números, os valores de:

- ❏ números de `unit`
- ❏ dimensões de *array*
- ❏ constantes

A vantagem de utilizar essa padronização é que os números são definidos somente em um único lugar, facilitando modificações.

Exemplos:

```
integer,   parameter :: c_w = kind(1.0)
real(p_w), parameter :: c_p_day = 1.0_p_w /(24.0_p_w * 3600.0_p_w)
real, parameter      :: c_kb = 1.3806504e-23
!# boltzmann constant [jk-1]
```

5.5. Arrays

Recomendada: Com arrays automáticos, potencialmente grandes, procure utilizar arrays alocáveis. Deste modo, caso não haja espaço suficiente durante a alocação do array na pilha, o programa não irá falhar.

Recomendada: Ao utilizar arrays alocáveis prefira dealocá-los explicitamente quando não forem mais necessários.

5.6. Allocatables versus pointers

Recomendada : É altamente recomendada a utilização de arrays alocáveis em vez de ponteiros. Arrays alocáveis são agora permitidos como componentes de estruturas nas extensões de 2003 e 95. 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 argumento reais necessitam sempre ter o ponteiro ou o atributo de destino (Ponteiros em C são o oposto).

5.7. Dimensões

Mandatória: Defina dimensionamento explícito ou arrays *assumed shape* (":").

Antes:

```
integer, dimension (*) :: array
!# Assumed size não é permitido
```

Depois:

```
integer, dimension (1:10) :: array1
!# Dimensão explícita
integer, dimension (:)    :: array2
!# Assumed shape é permitido
```

5.1 Arquivo de constantes físicas e numéricas:

Mandatória: Deve-se criar um único arquivo que englobe todas as constantes físicas e numéricas em padrão da linguagem fortran 90 ou superior, usando sempre a declaração de *'parameters'* em cada constante. Tal arquivo será usado como **'include'** em todos os códigos quando necessário.

Mandatória: Todas as variáveis devem usar nomes segundo a definição do padrão de nomes de variáveis que será definido mais abaixo e deve sempre iniciar com os caracteres **c_** . Para efeito de documentação deve-se usar a regra padrão de documentação definida em item posterior. Exemplo:

```
real, parameter :: c_pi=3.1415926
!# Pi value
real, parameter :: c_avogrado=6.022e+23
!# Avogrado number [mol^-1]
real, parameter :: c_bohrMagnetom=927.400915e-6
!# Magnetom de Bohr [J.T^-1]
```

5.3 Nomes de procedures:

Recomendada: As funções, programas, subrotinas, módulos ou qualquer procedure deve ter nome significativo que facilite entender sua funcionalidade.
Mandatória: Deve seguir as regras de *Camel Case* para os nomes de procedures e evitar nomes simples e curtos ou sem significados. Exemplos:

Inválido:

```
subroutine suba(a, b, c)
```

Válido:

```
subroutine computePotentialTemperature(temp, press, soilMoisture, sst)
```

5.4 Constantes de controle para dump:

Mandatória: Deve-se introduzir em cada procedure duas constantes que serão necessárias para manutenção de código e arquivos de *dump*. São elas: ***header*** e ***version***. *Header* deve conter o nome do *procedure* e *version* a revisão da mesma. Exemplo:

```
character(len=*), parameter :: header='computePotentialTemperature'
character(len=*), parameter :: version='1.0.1'
```

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

5.5 Encapsulamento em módulos:

Mandatória: Todos 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;
- Privada (*private*) - Somente visível dentro do módulo;

Mandatória: Os módulos devem conter suas próprias rotinas de inicialização se necessárias e especialmente aquelas para zerar variáveis. Considerar sempre que a criação de uma variável não a inicia para valor algum.

Mandatória: Os módulos não podem fazer uso (*use*) de outros módulos externos a ele, exceto nos casos de **includes** globais como o de constantes descrito acima em 5.1.

5.6 Drivers para módulos:

Recomendada: Se necessário devem ser criados drivers que adaptem o código principal e a dinâmica dos modelos a qualquer módulo encapsulado. Os drivers podem fazer uso de outros módulos através de *use* e tem a função de adaptação de estrutura de dados e unidades de medida entre os diferentes módulos.

Mandatória: O drive deve conter o nome que contenha o nome do módulo seguido da palavra *'Driver'*. Por exemplo, se um driver for criado para a radiação rrtmg (*radRrtm*) este deverá se chamar *radRrtmDriver*.

Deve-se atentar sempre para questões de performance de código, especialmente na criação de drivers. Geralmente drivers copiam estruturas de dados que causam cache miss devido a distância de memória. **Preferencialmente deve-se refatorar o código do módulo encapsulado** de forma que as estruturas de dados sejam compatíveis. Por exemplo: se no código principal do modelo a memória está definida nas dimensões (k,i,j) e no módulo encapsulado a memória está em (i,j,k) qualquer variável terá que ser convertida. Essa cópia gera um *overhead* no tempo de computação do modelo. Dependendo do tamanho do domínio soma-se ainda um custo por cache miss ou por page fault.

Mandatória: Os drivers criados devem chamar os procedures do módulo sempre por parâmetros e nunca por memória (*use*, etc).

5.7 Linhas em branco e tabulações/indentação:

Linhas em branco permitem que o código fique mais legível. Entretanto não há a necessidade de um número grande de linhas em branco entre linhas de código. Define-se então duas regras (O):

a)Duas linhas em branco:

- Entre seções importantes do código
- Entre definições de dados e sub rotinas dentro de um módulo

b)Uma linha em branco :

- Entre funções / subrotinas
- Dentro das funções / subrotinas, entre o escopo de variáveis e os blocos de código.
- Antes ou depois de linhas simples de comentário.
- Entre blocos lógicos dentro das funções / subrotinas.

Mandatória: Como regra geral para construção dos fontes, definiu-se a indentação com **2 espaços. NÃO** usar tabulação para esse fim, pois cada editor pode abrir a indentação de forma errada. Definir uma indentação deixa o código mais claro. Exemplo de uso:

Antes:

```
do j = 1,Jbmax
      do i=1,Ibmaxperjb(j)
      if (iPerIJB(i,j) == 123 .and. jPerIJB(i,j)== 56) then
            do k = KmaxInp,1,-1
            print *,  gPresInp(i,k,j),gUvelInp(i,k,j)
            print *, gPresInp(i,k,j),gVvelInp(i,k,j)
print *, gPresInp(i,k,j),gTvirInp(i,k,j)
```

```
            end do
 do k = Kmaxout,1,-1
   print *, gPresOut(i,k,j),gUvelOut(i,k,j)
            print *, gPresOut(i,k,j),gVvelOut(i,k,j)
            print *, gPresOut(i,k,j),gTvirOut(i,k,j)
         end do
  end if
      end do
   end do
```

Depois:

```
do j=1,Jbmax
  do i=1,Ibmaxperjb(j)
    if (iPerIJB(i,j) == 123 .and. jPerIJB(i,j) == 56) then
      do k=KmaxInp,1,-1
        print *, gPresInp(i,k,j),gUvelInp(i,k,j)
        print *, gPresInp(i,k,j),gVvelInp(i,k,j)
        print *, gPresInp(i,k,j),gTvirInp(i,k,j)
      end do
      do k=Kmaxout,1,-1
        print *, gPresOut(i,k,j),gUvelOut(i,k,j)
        print *, gPresOut(i,k,j),gVvelOut(i,k,j)
        print *, gPresOut(i,k,j),gTvirOut(i,k,j)
      end do
    end if
  end do
end do
```

5.8 Comandos por linha e tamanho de uma linha:

Mandatória: O número máximo de comandos por linha de código é 1. Não se deve usar ponto-e-vírgula e separar mais de um comando por linha .

Mandatória: Uma linha não pode ter mais de 80 caracteres de tamanho no total e deve se usar a continuação de linhas para os casos que excedem esse tamanho.

Mandatória:Para casos em que o conteúdo ao fim da linha é um string deve-se dividi-lo usando a concatenação de strings (//) e usar a continuação de linha (O).

5.9 Continuação de linha:

Usar o carácter que marca continuidade de linha abaixo ("&") nos seguintes casos (O):
Mandatória: Em fórmulas matemáticas: colocar o carácter de continuação "&" logo antes de um operador de matemática. Colocar o operador alinhado com o símblo de '='

na linha seguinte. Exemplo:

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

Mandatória: Havendo necessidade de quebrar linhas de argumentos de funções/subrotinas, colocar o carácter de continuidade de lina '&' logo antes de uma vírgula de separação de dado (",") e na linha seguinte começar com uma ',' alinhada com o primeiro carácter do nome do procedure. Exemplo:

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

Mandatória: Caso uma linha contenha um string que precise continuar na linha seguinte (citado em 5.8) 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. Exemplo:

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

5.10 Considerações especiais para end's if e do e laços rotulados:

Mandatória: Desvios condicionais com blocos if/end if e do/end do devem usar o end colapsado, isto é, usar endif e enddo ao invés de end if e end do (O).

Recomendada: Sempre que possível usar identificador (label) para os laços do indicando seu início e fim especialmente nos laços de muitas linhas(D). Exemplo:

```
outer: do iCount1=1,10
  do iCount2=1,10
    if ( iCount1 == 2  .and.  iCount2 == 3 ) cycle outer
    z(iCount2,iCoutn1) = 1.0d0
    if(iCount1>7) then
      Print *,'Just for test: ',iCount2
      Call adjustCount1Value(Icount1)
    endif
  enddo
enddo outer
```

5.11 declaração de variáveis e implicit:

Mandatória : Variáveis de entrada e saída devem estar declaradas com **implicit** adequado (*in, inout ou out)* e devem usar **explicit shape**.

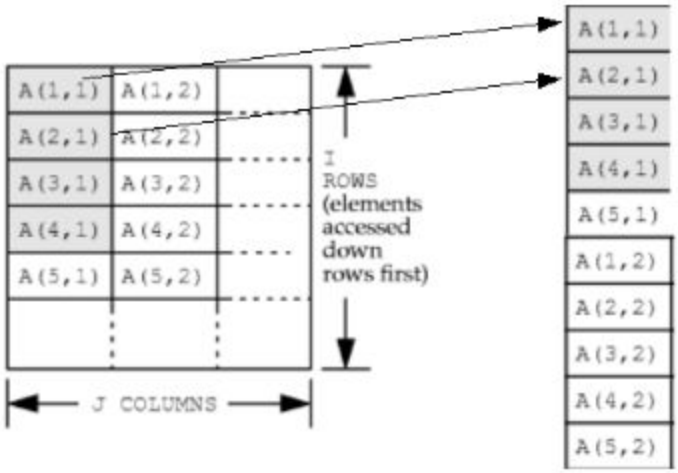
Mandatória: Todo procedure deve estar sob declaração de **implicit none**.

Mandatória: Para o caso dos módulos deve-se sempre considerar que todo o módulo é **private**, exceto as variáveis e procedures explicitamente declaradas como **public**.

5.12 Garantia de performance:

Recomendada: Faz-se necessário que regras básicas para melhoria de performance computacional devam ser usadas já na escrita do código.

Recomendada: O primeiro cuidado a se tomar é a preservação da ordem dos dados na memória ao se escrever laços no código. Diferente da linguagem C e suas similares o fortran monta o array linha a linha na memória numa mesma coluna.



Considerando que a memória cache dos computadores (processadores) são limitadas, é possível que buscas à cache não encontrem a posição de memória desejada. Isso causará um “cache miss” ou seja, uma nova carga da cache será necessária ocasionando uma baixa performance do programa. Para minimizar esses custos de acesso a memória é necessário que a ordem de busca aos elementos do array sejam adequadas. Percorrendo a memória na seqüência adequada o programa terá melhor performance. Assim é melhor estruturas de laços com o laço mais interno “por dentro”. Veja exemplo:

```
do jCount=1,2
  do iCount=1,5
```

```
    rCountPi(iCount,jCount)= c_pi*iCount
  enddo
enddo
do kCount=1,100
  do jCount=1,2
    do iCount=1,5
      bBountPi(iCount,jCount,kCount)= rCountPi(iCount,jCount)
    enddo
  enddo
enddo
```

Recomendada: Sempre que possível deve-se evitar uso de variáveis compartilhada por memória.

Mandatória: é proibido que se inicialize uma variável na sua declaração.

5.13 Uso de comandos descontinuados e clareza de código:

Mandatória: É proibido o uso de comandos descontinuados para fortran 90/95 como os listados abaixo:

- **pause**: Proibido devido a máquinas que usam submissão de jobs não o executarem sendo fonte de problemas de portabilidade;
- **equivalence**: proibido por ser fonte de invasão de memória;
- **common**: Deve ser substituído por módulos (memória). Seu uso como o equivalence é fonte de invasão de memória;
- **save**: Deve ser substituído por variáveis em módulos. Seu uso torna o código não thread safe e não puro impedindo os recursos de OpenMP;
- **data**: – Não deve ser usado. As variáveis devem ser inicializadas por subrotinas de inicialização em cada módulo. Em casos específicos de constantes usar parameter. O uso de data torna o código não thread safe e não puro impedindo os recursos de OpenMP.

Recomendada: Recomenda-se que não se use o comando **goto** ou **continue** sempre que possível substituindo-os por estruturas que usem condicionais e os comandos **cycle** e **exit**.(D)

Recomendada: Laços de decisão repetitivas do tipo **if**, **then**, **elseif** e **else** sempre que possível devem ser substituídos por estrutura de case (D). Veja exemplo:

```
Select case (iosError)
  case(:900)
    dumpError(header,version,'Unknown error')
  case(913)
```

```
    dumpError(header,version,'Out of Free Space')
  case(963:971)
    dumpError(header,version,'Format error')
  Case default
    dumpError(header,version,'Miscellaneous error')
End select
```

6. Documentação de código

Para facilitar o entendimento do código e torná-lo um código com maior usabilidade, portabilidade, manutenibilidade e confiabilidade é necessário que o mesmo seja adequadamente documentado internamente nas linhas do próprio código.

Escolheu-se uma ferramenta desenvolvida com esse propósito, Ford, desenvolvida por Phd. C. MacMackin de Oxford e tendo código original disponível em [[https://github.com/cmacmackin/ford]] conforme citado em 4.3. Para que o sistema de documentação funcione é necessário que sejam escritas tags em formato específico no interior do código de forma a que o documentador possa extrair as informações do mesmo.

Todo código fortran, como é o caso dos modelos adotados no CGCPT, possui em seu corpo procedures específicas à saber:

- program
- subroutine
- function
- module

Cada procedure deve possuir um cabeçalho que a descreva adequadamente. Espera-se que definam ao menos sua funcionalidade, autor e data de criação e estejam identificadas suas variáveis com o significado de cada uma. No modelos acrescentamos ainda outras necessidades tais como: informação de bugs, listas de tarefas necessárias (todo lists), atualizações(changes) e informações de copyright ©.

6.1 Padrão básico de cabeçalho

As recomendações para o cabeçalho para cada procedure são as seguintes:

- Toda linha do cabeçalho de documentação deve começar com os caracteres **!#**
- Logo após a declaração do programa (program), função (function), subrotina (subroutine) ou módulo (Module) acrescentar a primeira linha de cabeçalho com uma breve descrição da função do procedure começando com os caracteres de documentação (O), por exemplo:

```
subroutine aerSaltFlux(flux,w10,sst)
!# Obtain the seasaltflux and mass over waves cap (surface)
```

- Logo após abrir uma nota para descrever a funcionalidade, autor e data de criação com a tag @note e terminando com @endnote (O)
- Entre as tags de nota, usando os caracteres **Detail:** descrever a funcionalidade o mais precisamente possível. Deve-se usar se necessário figuras,fórmulas, links, etc (O).
- Entre as tags de nota, usando os caracteres **Author:** informar o nome do autor e, se possível, o seu email (O).
- Entre as tags de nota, usando os caracteres **Date:** Colocar a data de criação do arquivo fonte ou do procedure (O).
- Preferencialmente colocar 3 traços entre 'detail','author' e 'Date' nas linhas de cabeçalho. Usar sempre os caracteres indicativos de documentação (D).
- Fechar o setor de nota com @endnote
- Veja abaixo um exemplo de como ficaria um cabeçalho de subrotina usando a documentação:

```
subroutine aerSaltFlux(flux,w10,sst)
!# Obtain the seasalt flux over waves
!#
!# ![](http://logo.do.projeto/logo.png "")
!#
!# @note
!#
!# **Detail**: This subroutine solves the flux of seasalt aerosol in each specific bean
!#---
!# **Author**: Rodrigues, L.F. **&#9993;**<mailto:luiz.rodrigues@inpe.br>
!#---
!# **Date**: 2015Jul
!# @endnote
```

6.2 Padrão Avançado de cabeçalho

Após aplicar o cabeçalho conforme mostrado no tópico anterior recomenda-se que se informe bugs, alterações posteriores ao código original, listas de alterações e informação de copyright. Se o código está sob licença CC-GPL ou outra licença livre isso deve ser identificado e se partes são proprietárias as mesmas devem estar identificadas e citadas. Abaixo o mínimo que deve ser usado(O):

- **Alterações de código**: devem estar entre as tags @changes e @endchanges e cada linha com alteração deve se iniciar com um sinal de +, o autor, a data de alteração e uma descrição sucinta da alteração realizada. O sistema de controle de versões do modelo mantém o histórico de alterações e pode-se voltar (rollback) para versões

anteriores. Entretanto é saudável e recomendado que se faça uma anotação das modificações no cabeçalho do próprio código.

- **Bugs**: Bugs conhecidos e ainda não solucionados que caso sejam detectados ou informados por terceiros devem estar identificados entre as tags @bug e @endbug da mesma maneira que se faz com @changes. Caso não haja bug reportado deve-se declarar com a informação 'No active bugs reported until now'. A informação de bug, se houver, deve ser retirada do cabeçalho quando o mesmo for resolvido.

- **Novas funcionalidades**: Caso haja a necessidade de uma nova funcionalidade ou alteração futura do código (não causada por bug) recomenda-se que se identifique entre as tags @todo e @endtodo da mesma forma que reportado acima.

- **Avisos e licenças**: Por fim deve-se mostrar um aviso com a licença usada e, se pertinente, com um link que aponte para a mesma bem como qualquer aviso importante relativo ao código. Para isso deve-se colocar o texto entre as tags @warning e @endwarning.

  Vejamos um exemplo abaixo:

```
!# @changes
!#
!# + The ammount of sea salt mass was multiplied for 1K to adjust
!# @endchanges
!#
!# @bug
!# + The routine not work with wind upper 100m/s
!# @endbug
!#
!# @todo
!#  + adapt to new chemical and aerosol subroutines &#9744; <br/>
!# @endtodo
!#
!# @warning
!# Now is under CC-GPL License, please see
!# &copy; <https://creativecommons.org/licenses/GPL/2.0/legalcode.pt>
!# @endwarning
!#
!#---
```

6.3 Declaração para variáveis e constantes.

Todas as variáveis de um módulo, subrotina, função ou programa principal e todas aquelas passadas entre chamadas devem estar devidamente documentadas. Assim também deve ser feito para todos os parâmetros (constantes) utilizados em um código. Descrevê-las é importante para o bom entendimento do escopo e de sua função dentro do código além de seu uso adequado. Para isso, na linha imediatamente abaixo da declaração da variável, deve-se colocar o cabeçalho para a mesma. Pode-se usar múltiplas linhas para descrever a variável (O). Caso a variável ou constante tenha uma dimensão física ou unidade de medida a mesma também deve ser mostrada entre colchetes na documentação (O). Vejamos um exemplo:

```
implicit none

real, intent(in) :: wind10
!# Wind at 10 m [m/s]
!# this is a composit wind U and V.
 real, intent(in) :: sst
!# Sea surface temperature [K]
real, intent(out) :: flux(3)
!# Flux of aerosol #/m2s

Integer :: kCount
!# Column counter

 do kCount=1,nbins
     flux(kCount)=3.84e-06*(aFactorK(kCount)*sst+bFactorK(kCount))*wind10**3.41
enddo
```

6.4 Uso de caracteres Fórmulas, HTML e Markdown.

O Ford aceita caracteres de HTML bem como o padrão markdown. Nos exemplos acima observa-se que alguns caracteres em HML são usados: ✉ para o símbolo de uma carta (✉), ☐ para o símbolo de 'check' (☐) e © para o símbolo de copyright (©). Qualquer outro símbolo na notação HTML pode ser usado. Símbolos em HTML podem ser consultados em http://erikasarti.com/html/dingbats-simbolos-desenhos/

Além dos símbolos o Ford é compatível com tags HTML como a
 usada nos exemplos acima. Algumas tags HTML são mais simples quando usadas através do padrão markdown. Acima nos exemplos vemos usar algumas dessas tags:

- ** (dois asteriscos) colocados antes e depois de uma palavra indicam negrito (um asterisco é itálico)
- + indica um item de lista
- <> textos entre sinais maiores e menores são links para páginas externas

- ; É um link para uma figura no endereço http web definido

Mais tags markdown podem ser vistas em https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet.

Figuras podem ser acrescentadas à documentação com a tag específica mostrada acima (6.1). É fundamental que a figura esteja disponível on-line em algum sítio na internet. Usa-se os parênteses para indicá-lo.

6.4 Fórmulas e equações na documentação.

O Ford admite o uso de padrões de matemática do LATEX. Para tanto deve-se usar o mesma nomenclatura padrão dos textos em LATEX sem se esquecer que deve estar iniciado com os caracteres de documentação, !#. Deve-se usar as definições de início de fórmula matemática, seja pelo uso de [\ ]\ como delimitadores ou \begin{equation} \end{equation}. Por exemplo:

```
MODULE seaSalt
  !# Module to determine the whitecap seasalt flux and mass over waves (surface)
  !#
  !# @note
  !#[](http://logo.do.projeto/logo.png "")
  !#
  !# **Detail**: The module is based on paper *Modeling sea-salt aerosol in a coupled climate and     !#sectional
  !# microphysical model: mass, optical depth and number concentration*, Atmos. Chem. Phys., !# 4610, 2011 - T. Fan and O. B. Toon, That determine the whitecap flux and mass of sea salt particles over (on surface) the
  !# sea waves.
  !# \[F_n=3.84 \text{x}10^{-6}  (ak_n  sst+bk_n) U_{10}^{3.41}\]
  !# and
  !# \[M_n=F_n*M_{{aer}_n}*1.0\text{x}10^{-3}\]
  !# Where: <br/>
  !# \[sst \text{ = sea surface temperature [k]}\]
  !# \[U_{10} \text{= wind at 10 m [m/s]}\]
  !# \[M_{{aer}_n} \text{= Aerosol particle mass in each bin}\]
  !# \[ak_n \text{=parameter factor for equation (a)}\]
  !# \[bk_n \text{=parameter factor for equation (b)}\]
  !# \[n \text{= especific bin of aerosol [1,2 or 3]S}\]
  !# ---
  !# **Documentation**: <http://siteDoModelo.com/>
  !# ---
  !# **Author**: Rodrigues, L.F. **&#9993;**<mailto:luiz.rodrigues@inpe.br>
```

```
!# ---
!# **Date**: 2015Jul
!#
!# @endnote
!#
!# @changes
!#
!# +
!# @endchanges
!# @bug
!# No active bugs reported now
!# @endbug
!#
!# @todo
!#  &#9744; <br/>
!# @endtodo
!#
!# @warning
!# Now is under CC-GPL License, please see
!# &copy; <https://creativecommons.org/licenses/GPL/2.0/legalcode.pt>
!# @endwarning
!#
!#--- ----------------------------------------------------------------------------------
```

Como resultado as equações aparecerão na documentação como na figura abaixo:

$$F_n = 3.84x10^{-6}(ak_n sst + bk_n)U_{10}^{3.41}$$

e

$$Mass_n = F_n * M_{aer_n} * 1.0x10^{-3}$$

6.5 Automação para inserção de cabeçalho.

Um bom editor para código fortran é o atom. Para quem usa o editor Atom (veja em https://atom.io/) foi criado um snippet especial para os cabeçalhos. Ele automatiza a confecção dos mesmos. Basta digitar os caracteres head e pressionar a tecla tab que um cabeçalho é colocado automaticamente na posição do cursor facilitando a edição de código. Mas para isso é preciso criar um 'snippet' no Atom e introduzir o código do mesmo.

Antes de editar o snippet é necessário que o pacote de fortran esteja instalado. Caso não esteja, para instalar o pacote de fortran, estando com o editor aberto, basta clicar em **'Edit'** no painel superior e depois em **'preferences'**. Clique então em **+install** no menu à esquerda/centro. Digite fortran na caixinha onde se lê 'search packages' e veja a lista de pacotes fortran que irá aparecer. Escolha 'language-fortran' e clique em **Install** (tecla azul). Aguarde a instalação e pronto.

Para editar o snippet, estando com o editor aberto, basta clicar em **'Edit'** no painel superior e depois em **'preferences'**. Após isso clique em **packages** no menu à esquerda/centro e aparecerá o pacote **language-fortran** na tela à direita. Clique então em **settings** na janela de **language-fortran**. No menu totalmente à esquerda da tela irá aparecer a estrutura com a palavra snippets. Abrindo a aba aparecerá a configuração dos snippets para fortran, chamado **language-fortran.cson**.

Clicando duas vezes sobre ele aparecerá o código na janela de edição. Procure as palavras abaixo (use Control-f):

```
'Write':
   'prefix': 'write'
            'body': 'write(unit=${1:iounit}, fmt="(${2:format string})", iostat=${3:ios}${4:, advance=\'NO\'}) ${5:variables}\nif ( ${3:ios} /= 0 ) stop "Write error in file unit $1"$0'
```

Logo após esse trecho, na linha seguinte, inclua o seguinte texto:

```
'header':
   'prefix': 'head'
   'body': '!# put here a brief description\n
   !#\n
   !# @note\n
   !# ![](http://sitedoprojeto/logo.png "")\n
   !#\n
   !#**Detail**: put here a detailed description\n
   !# remember: You can use formulas in Latex format like: \n
   !# \[F_n=3.84 \text{x}10^{-6}  (ak_n  sst+bk_n) U_{10}^{3.41}\] \n
   !# or You can use figures or graphs available in internet like: </br>\n
   !# ![](https://image.flaticon.com/icons/svg/44/44386.svg "")\n
   !# or any others HTML and Markdown symbol or tag.\n
   !# \n
   !#---\n
   !#**Documentation:** <http://brams.cptec.inpe.br/documentation/>\n
   !#---\n
   !#**Author**: Put here the author name and email in <mailto:youremail@yoursite>\n
   !#---\n
   !#**Date**: put here the date of creation\n
   !#\n
```

```
!# @endnote\n
!#\n
!# @changes\n
!# \n
!# + if there are changes in code put the information of changes there \n
!# + use one plus signal for each one \n
!# @endchanges\n
!# @bug \n
!# + if ther are actives bugs put the bugs info here. \n
!# + if not, put the message: No active bugs reported now \n
!# @endbug  \n
!# \n
!# @todo \n
!# + if there are some new feature(s) to do, please, put it here &#9744; <br/> \n
!# @endtodo \n
!# \n
!# @warning \n
!# put here information about warning, included the licenses \n
!# Now is under CC-GPL License, please see \n
!# &copy; <https://creativecommons.org/licenses/GPL/2.0/legalcode.pt> \n
!# @endwarning \n
!# \n
!#--- \n
'
```

Salve o arquivo, saia e pronto. Agora é só abrir o diretório e seu fonte e na linha que desejar colocar o cabeçalho digite **head** e tecle **tab**, o template do cabeçalho será colocado para que seja preenchido com os dados corretos.

Caso não use o atom recomendamos que crie um template com o cabeçalho e o use sempre que precisar. Um exemplo está logo abaixo.

```
!# put here a brief description
!#
!# @note
!# ![](http://brams.cptec.inpe.br/wp-content/uploads/2015/11/logo-brams.navigation.png "")
!#
!#**Detail**: put here a detailed description
!# remember: You can use formulas in Latex format like:
!# \[F_n=3.84 \text{x}10^{-6}  (ak_n  sst+bk_n) U_{10}^{3.41}\]
!# or You can use figures or graphs available in internet like: </br>
!# ![](https://image.flaticon.com/icons/svg/44/44386.svg "")
!# or any others HTML and Markdown symbol or tag.
!#
!#---
!#**Documentation:** <http://brams.cptec.inpe.br/documentation/>
!#---
!#**Author**: Put here the author name and email in <mailto:youremail@yoursite>
!#---
```

```
!#**Date**: put here the date of creation
!#
!# @endnote
!#
!# @changes
!#
!# + if there are changes in code put the information of changes there
!# + use one plus signal for each one
!# @endchanges
!# @bug
!# + if ther are actives bugs put the bugs info here.
!# + if not, put the message: No active bugs reported now
!# @endbug
!#
!# @todo
!# + if there are some new feature(s) to do, please, put it here &#9744; <br/>
!# @endtodo
!#
!# @warning
!# put here information about warning, included the licenses
!# Now is under CC-GPL License, please see
!# &copy; <https://creativecommons.org/licenses/GPL/2.0/legalcode.pt>
!# @endwarning
!#
!#---
```

7. Testes autônomos e modulares

Os códigos devem implementar sempre que possível, dentro do próprio módulo, testes autônomos que garantam seu funcionamento no modo stand-alone. Para isso, se necessário, deverá ter arquivos de entrada para o teste e arquivos de saída para que possa ser feito comparação além de um código que permita executar o teste.

8. Escalabilidade e eficiência

Todos os códigos devem implementar garantias de thread safe, serem puros e garantirem uma boa escalabilidade dentro dos padrões da máquina. Preferencialmente devem garantir maior eficiência na escalabilidade de que códigos com a mesma função já utilizados pelo CGCPT.

9. Garantia de portabilidade

Qualquer código desenvolvido no CGCPT deve garantir uma portabilidade mínima. Para tanto a linguagem deve obedecer a alguns padrões e requisitos mínimos (O):

a) Ser adequado ao padrão Fortran 95 (ISO/IEC 1539-1:1997) sem usar as funções desativadas (deprecated)

b) Ser capaz de compilar em ao menos 3 compiladores: Intel, Portland PGI e GNU Suíte (Gfortran e GCC)
c) Deve ser capaz de rodar em máquinas disponíveis no CGCPT e ser adequado a arquitetura das mesmas e em cluster massivamente paralelos.

10. Termos para aceitação de modelos e códigos externos

É saudável a introdução de códigos atualizados e modernos nos sistemas e modelos em uso ou desenvolvidos pelo CGCPT. É esperado que, com as regras implantadas, os modelos em uso no CGPTC, na fase de desenvolvimento, pre-operacionais ou operacionais estejam adequados a essas regras. Portanto a importação de códigos para esses modelos devem estar sob critérios específicos. São eles:

a) Não comprometer a qualidade do software (modelo) já implementada no CGCPT;
b) Se o mesmo substitui uma parametrização usada por outra nova, este deve estar com um mínimo de performance, maior ou o mais próximo possível, da parametrização em uso;
c) Deve conter documentação dentro das regras desse documento ou permitir a introdução das mesmas em curto prazo - deve-se definir pessoa ou equipe para a tarefa.
d) O grupo dmdcc fará uma análise do código a ser utilizado e utilizará um critério de avaliação envolvendo os itens do padrão de código (Code Patterns - 5) adotado, documentação e testes (6 e 7) e performance (eficiência, escalabilidade, pureza - 8). Para cada item será atribuída uma nota de zero a 4, sendo:
    i) 0 - não atende o item
    ii) 1 - atende menos de 50% do exigido no item
    iii) 2 - atende 50% do item
    iv) 3 - atende mais de 50% item
    v) 4 - atende completamente (feito dentro das normas exigidas)
e) Códigos com menos de 50% de conformidade tem custos altíssimos de retrabalho. Podem ser usados em modo de pesquisa mas deve-se tomar o máximo cuidado ao aceitá-los como código do CGCPT.
f) Códigos com conformidade maior ou igual a 50% serão aceitos e são passíveis de retrabalho.
g) Para códigos com aceitação menor que 100% dos pontos será feita uma análise do trabalho necessário para a conversão. Dessa análise resultará 3 valores para apresentação, sendo:
    i) Dificuldade de implementação (fácil=1, mediana=2,difícil=3)
    ii) Risco na implementação (Baixo=1, médio=2, alto=3)

iii) Força de trabalho em Homens/Hora (estimado) para a adequação

h) O grupo desenvolverá metodologia própria para análise de cada necessidade e cada item terá uma descrição da motivação da nota.

i) Será produzido um relatório padrão contendo 3 documentos:
   i) Avaliação dos itens de padronização, documentação e performance do código
   ii) Análise de implementação, riscos e mão de obra
   iii) Relatório com avaliação geral do código.
   Os 3 documentos estão em uma planilha que pode ser obtida no endereço
   https://drive.google.com/open?id=1iwZvIAuim15tONzUaLVDuWOioMsI9guL

   O máximo de pontos positivos - conformidade do código - é de 96. Para os pontos negativos temos os de Dificuldade de implementação e o de riscos, 66 pontos cada.

   Esses valores e os de homens/hora para a implementação devem ser considerados na decisão de utilizar o código.

11. Referências

- *https://www.devmedia.com.br/ciclos-de-vida-do-software/21099*
- *https://www.devmedia.com.br/extreme-programming-conceitos-e-praticas/1498*
- *SOMMERVILLE, Ian, Engenharia Software. Addison Wesley. 8ª ed*
- *PRESSMAN, Roger, Engenharia Software, McGraw-Hill. 6ª ed*
- *Case Maker Inc., What is Rappid Application Development?*
- *PISKE, Otavio. SEIDEL, Fabio, Rapid Application Development*
- *Norma NBR ISO/IEC 12207:1998*
- *SPINOLA, Rodrigo, Boas Práticas de Engenharia de Software, 2011*
- *PFLEGER, Shari, Engenharia de Software – Teoria e Prática, Prentice Hall, 2ª Ed*
- *PAULA Filho, Wilson, Engenharia de Software: fundamentos, métodos e padrões, LTC, 3ª Ed*
- *JENKINS User Documentation. https://jenkins.io/doc/* . Acesso em: 21 set. 2018.