Guidelines for COHERENS developers

Basic rules

The basic rules for programming can be summarised by a few key words

- **Portability**
    - FORTRAN 90 ANSI standard to make the code compiler-independent.
    - Ensure maximum independency of computing platform.
        - COHERENS is intended for implementation on UNIX/LINUX systems (WINDOWS, although not excluded *a priori*, is not supported)
        - Use *bash* mode for shell scripts.
        - Generic *Makefile* independent on type of compiler and computing platform.
        - System dependent parameters, such as path names for the location of libraries should be provided from a user-defined configuration file.
- **Error-free**
- **Optimisation**
    - Optimise (minimise) computing time
    - Optimise (minimise) internal memory use (provided this is not in conflict with CPU optimisation)
- **Efficiency**
    - Use generic formats (i.e. for parallel communications, I/O, array interpolation, routines used by different model compartments,...).
    - When a new development is added to the code, make a clear distinction to what is new and should be implemented by creating new routines and files, and what can be integrated within the already existing routines and files.
    - Further recommendations are listed below.
- **Transparancy**
    - The code should be written in a transparant user-friendly readable format so that it is understandable by other developers even if this implies a (slightly) less optimum code.
    - Use standard "COHERENS" coding guide lines (see Chapter "Program conventions and techniques" in the User Documentation and the text below.
    - Compliance with the User Documentation.
    - Standard (consistent) formats for internal documentation (see below).

Priorities

- Portability and an error-free (completely debugged) code must obviously have the highest priority.
- Optimisation has a higher priority than transparancy (and even efficiency) for those parts (routines) of the code requiring a "significant" computing time (e.g. routines dealing with advection/diffusion or transports of scalars/momentum).
- Transparancy has a higher priority than optimisation for routines with less impact on CPU time, such as routines which are only called once like initialisation routines.
- Important to note is that users of COHERENS must be able to apply the code without knowledge of its internal structure. This implies that transparancy (user-friendlyness) has a higher precedence than code efficiency and even optimisation for those aspects of the code related to model setup (which is not always easy in practice).

Files

A convention is adopted for the names of the files used by the COHERENS program (FORTRAN source files, files for compilation and scripts) so that their purpose can be derived from their name.

- FORTRAN source code files:
    - MODULE files with declaration of "global" variables (i.e. variables used in more than one routine). The name of the file consists of lower case characters without '_' *(underscore)* followed by the suffix *.f90,* e.g. *currents.f90.*
    - MODULE routines files containing module routines (their purpose and use is explained below). File name consists of a prefix with lower case characters containing an '_'*.* Suffix is *.F90* if the file is passed to the CPP preprocessor before compilation (e.g. the file contains *#ifdef* statements). Otherwise the suffix is *.f90*. For example, *paral_comms.f90*, *grid_interp.F90* .
    - Files with external (as defined by the ANSI standard) routines. Suffix is *.F90* for files needing preprocessing, *.f90* otherwise.
    - *Usrdef_** files used for setup by the user. Suffix is *.f90* always (i.e. *#ifdef* statements are not allowed).
    - Some consistency in names is recommended:
        - *modvars_routines.f90, sedvars_routines.f90, biovars_routines.f90*: routines where variable and forcing file attributes are defined for respectively physical, sediment, biological variables
        - *physpars.f90, sedpars.f90, biovars.f90*: modules with declaration of (scalar) parameters for the physical, sediment, biological model
        - *model_output.f90, sediment_output.f90, biology_output.f90*: routines for standard output of physical, sediment, biological variables
        - *Density_Equations.F90* contains routines related to density calculations (temperature, salinity, equation of state, ...), *Hydrodynamic_Equations.F90* routines for the hydrodynamics (currents and surface elevation), *inout_routines.f90* for

        standard input/output, *time_routines.f90* for time and calendar routines.
    - ...
    - The name should refer, as exactly as possible, to the purpose of its containing routines. This means that routines in the file whose purpose is not related to the file name, should be moved to another or a new file. Only possible exception may be some utility routines only used by routines within the file.
- Files for compilation:
  - All files have standard names which can only be changed after concertation with MUMM and (eventually) other developers. The name *Makefile* is standard and will not be changed at all !
  - The name of the dependency file for the "physical" core component of COHERENS is *dependencies_cmp*. Other dependency files have names of the form *dependencies_** (where * is *sed, bio*, ...).
- Standard COHERENS forcing and output files. Names are defined by COHERENS and can only be changed as part of a developers task related to I/O.
- There are no conventions for external (non-standard COHERENS) data files or scripts created by users for specific applications.

Compilation

- The contents of *Makefile* should not be changed, except as part of specific development tasks intended to change the procedures for installing and compiling COHERENS. In that case, all changes in *Makefile* must be generic (i.e. platform and compiler independent). All compiler dependent information (like compiler options) is obtained from *compilers.cmp* and all system dependent information from the COHERENS configuration file.
- All developers are free to add new targets in *compilers.cmp*, provided they are useful for other users as well. It is recommended to add a target for debugging for each compiler.
- The name of the configuration file (where e.g. pathnames of external libraries and CPP compiler options are defined), is *coherensflags.cmp*. Its contents are platform dependent. It is recommended to keep the default (empty) version unchanged in the *comps* directory of COHERENS. Each user can define its own version (with the same name) outside the COHERENS directory and outside the svn versioning system. Anyway, it is strongly forbidden to upload a user version to the svn repository. Note that the concept of a configuration file may be changed in a future version as part of a developers task.
- Each time a developer creates a new revision, he/she has to check first the compilation dependencies and make the necessary changes in the dependency files. These dependencies are defined as follows:
  - A file A depends on file B, if file A contains a USE statement of a module contained in file B. I admit that this definition is not complete since a call from a routine in file A to an external routine located in B, also makes file A dependent on B. Checking of these dependencies is not performed during compilation. The reason is that it would make the dependency files much more complicated and can lead to errors in the dependency structures (i.e. recurrent dependencies) which are not easy to detect and may give strange error messages during compilation.
  - The following dependency hierarchy is adopted in the dependency files:
    - Module files *syspars.f90* and *datatypes.f90* have no dependencies at all.
    - All other module files can only depend on those two module files.
    - Module routine files can depend on all module and all other module routine files. Care has to be taken by the developer that no (hidden) recurrency occurs. For example, if file A depends on file B and file B depends on file C, file C cannot depend on file A.
    - Files with external routines may depend on all module and module routine files, but not on files with external routines.
    - *Usrdef_** files are assumed to depend on all module and module routine files.
  - The order of dependencies on a line in the dependency file is, in principle, irrelevant. For transparancy, the ordering is taken as follows: name of the object file, name of the source code file, name of the module object files on which the file depends in alphabetical order, names of the module routine object files on which the file depends in alphabetical order, e.g.

```
math_library.o: math_library.F90 grid.o gridpars.o iopars.o \
                switches.o syspars.o array_interp.o \
                error_routines.o time_routines.o
```

Subversion

Only a few comments are given here. Detailed information about the use of svn can be found in the subvsersion manual.

- It may occur that, when a file is downloaded from the server, '^M' characters appear on the end of lines. They arise from copying the file from a Windows to a Linux system. The reverse is even worse since Windows does not recognise the line endings of a Linux file. The problem can be avoided by setting the following svn property to each new file:

    svn propset svn:eol-style native file_name

- In addition, the Date and Revision properties have to be set on each new file

    svn propset svn:keywords "Date Revision" file_name

- Each new file must be added to the revision system by

    svn add file_name

- A file is removed from the revision system by

```
svn delete file_name
```

- It is recommended that developers remove code used for debugging before committing, unless some agreement is made with the other members of the developer's group and the code is committed as a working revision on a local branch.
- Writing to the trunk branch on the svn COHERENS repository is strictly forbidden except by the administrator (currently Patrick Luyten).

General programming conventions

- All source code (except *#ifdef* statement blocks) are written in FORTRAN 90 using the "free format" style:
    - Program lines start at column 1, with exception of statements within control structures such as IF and WHERE block, DO loops and SELECT CASE constructs which are intended by 3 spaces to the right. Lines within a nested construct are intended with respect to the previous one. For clarity, no indentation is applied if the DO/ENDDO statement of a nested loop is located below/above the DO/ENDDO statement of the parent loop. For example,

```
idesc_360: DO idesc=1,MaxIOTYpes
ifil_360: DO ifil=1,MaxIOFiles
   SELECT CASE (idesc)
      CASE (io_mppmod,io_modgrd,io_metgrd,io_sstgrd,io_biogrd,&
          & io_nstgrd,io_biospc,io_rlxobc,io_nstspc)
         modfiles(idesc,ifil,:)%nocoords = 0
      CASE (io_1uvsur,io_2uvobc,io_3uvobc,io_salobc,io_tmpobc,&
          & io_bioobc)
         IF (ifil.EQ.1) THEN
            modfiles(idesc,ifil,:)%nocoords = 0
         ELSE
            modfiles(idesc,ifil,:)%nocoords = 1
         ENDIF
      CASE (io_inicon,io_2uvnst,io_3uvnst,io_salnst,io_tmpnst,&
          & io_bionst,io_metsur,io_sstsur,io_biosur)
         modfiles(idesc,ifil,:)%nocoords = 1
   END SELECT
ENDDO ifil_360
ENDDO idesc_360
```

    - Comments start with a '!' in the first column. Although allowed by the FORTRAN 90 standard, the common practice is, for reason of clarity, not to use comments after the first column (i.e. in the middle of a line).
    - Short statements may be written on one line by inserting a ';' before the next statement.
    - Statement labels always start at the first column.
- Although the free format allows line lengths upto 132 characters, a length of 80 characters is taken for clarity since this avoids scrolling of code text on standard (non-maximised) X-windows used on UNIX/LINUX machines or when model code is sent to a printer.

Internal documentation

- Each source code file, routine and module must have a header in a specific COHERENS format. There are, of course, plenty of examples in the code.
    - If a file contains more than one routine, a general header is placed on top of the file:

```
!************************************************************************
!
! *Sediment_Equations* COHERENS sediment model
!
! Author - Alexander Breugem and Boudewijn Decrop (IMDC)
!
! Version - @(COHERENS)Sediment_Equations.F90  V2.5
!
! $Date: 2013-03-18 10:36:17 +0100 (Mon, 18 Mar 2013) $
!
! $Revision: 535 $
!
! Description - The model contains the equations for calculating suspended
!               sediment transport and bed load, including
!               parameterisations of important processes such as
!               settling velocities
!
! Routines - ackerswhite_params, bartnicki_filter, bed_slope_arrays,
!  beta_factor, diff_coef_waves, equilibrium_concentration,
!  equilibrium_timescale, median_particle_diameter, sediment_advdiff,
!  sediment_bedload, sediment_equation, sediment_suspendedload,
!  sediment_totalload, settling_velocity, thetastar_engelund_hansen
!
!************************************************************************
!
```

- Don't forget to write the name(s) of the author(s) which forms part of the license agreement. Company or institute can (optionally) be added in parentheses after the author's name.
- The date and revision information are inserted by subversion automatically after committing, provided the Date and Revision properties are set for the file (see above). When the file is created, only the names of the fields must be inserted:

```
! $Date$
!
! $Revision$
```

- A list of containing routines, in alphabetical order,is given in the "Routines" header field.
- The version number (V2.5 in this case) denotes the most recent COHERENS version where changes have been made in the file. Developers may use V2.x in case the final version number is not yet known.

- The header used for each routine within the file has the following format:

```
SUBROUTINE transport_at_C_3d(psic,source,vdifcoefatw,iopt_adv,iopt_hdif,&
                           & psiobu,psiobv,iprofrlx,ibcsur,ibcbot,&
                           & nbcs,nbcb,bcsur,bcbot,ivarid)
!************************************************************************
!
! *transport_at_C_3d* Solve advection-diffusion equation for a
!                     3-D quantity at C-nodes
!
! Author - Patrick Luyten
!
! Version - @(COHERENS)Transport_Equations.F90  V2.5
!
! Description -
!
! Reference -
!
! Calling program - salinity_equation, temperature_equation
!
! External calls - relaxation_at_C, Xadv_at_C, Xcorr_at_C,
!                  Xdif_at_C, Yadv_at_C, Ycorr_at_C,
!                  Ydif_at_C, Zadv_at_C, Zcorr_at_C,
!                  Zdif_at_C
!
! Module calls - error_alloc, exchange_mod, num_halo, tridiag_vert
!
!************************************************************************
!
```

    - Note that all listing is in alphabetical order.
    - The Date and Revision fields are absent, since svn can only insert them at one place in the file.
    - The FORTRAN declaration of the routine comes before the header.
    - Lines with empty fields may be omitted.
  - The format for module files is

```
MODULE switches
!************************************************************************
!
! *switches* Model switches
!
! Author - Patrick Luyten
!
! Version - @(COHERENS)switches.f90  V2.5
!
! $Date: 2013-02-18 10:20:56 +0100 (Mon, 18 Feb 2013) $
!
! $Revision: 533 $
!
! Description -
!
!************************************************************************
!
```

- Each variable in a module file, routine argument and (optionally) local variable is documented internally by its name, type, description and unit (if any). The order for documentation is alphabetically for modules (and local variables) or the order in which they appear within the routine declaration statement in case of arguments.
- Each part of the code in a routine is divided in sections, subsections, ... The section numbers and names are given by comment lines. For more information about the layout of routines in COHERENS, see Section 6.1 of Chapter "Program Conventions and techniques" of the User Manual.
- Developers may add additional comment lines, which for readabilty should start with a '!' in the first column.

Name syntax

- **Program variables**
  - Use lower case only. Exceptions are system variables as defined in *syspars.f90* (e.g. *MaxIOFiles*). These parameters can only be changed by developers. Underscore characters are not recommended except for reasons of clarity. For example, *bstresatc_sed* denotes the bottom stress at C-nodes as used in the sediment model to make it distinct to *bstresatc* as used in the

hydrodynamics. Numbers should be avoided for variables with a global scope.
- FORTRAN 90 specific name definitions or key words are always in upper case, such as IF, RETURN, SUBROUTINE, ALLOCATE, REAL, ... This applies also for the names of FORTRAN 90 intrinsic routines.
- Variables should refer to their meaning for clarity. This applies in the first place for global variables (declared in a module). It is recommended to supply a name similar to the one in the User Documentation (e.g. *zeta* denoting the surface elevation).
- COHERENS does not use implicit typing rules (see below) which means that all variables need to be declared. However, it is recommended to use for convenience
  - names starting with i to n for integer variables
  - names starting with a-h or o-z for all other variables (real, character, logical, complex, double)
- Names of global variables should not too short (min. 4 characters) do avoid duplication, and too long (more than 10 characters) except for reasons of clarity. In particular, names like *a*, *x*, *uc*, *sed* are not accepted.
- Local variables should, as much as possible, have names referring to their meaning. This may not be the case for auxiliary variables. No specific convention exists for auxiliary scalar variables. For arrays, the following conventions are used:
  - *array2dc* for 2-D arrays at C-nodes (or *array2dc1*, *array2dc2*, ... in case more than one array is needed)
  - *array3dc* for 3-D arrays at C-nodes (or *array3dc1*, *array3dc2*, ...)
  - *array2du* for 2-D arrays at U-nodes (or *array2du1*, *array2du2*, ...)
  - *array3du* for 3-D arrays at U-nodes (or *array3du1*, *array3du2*, ...)
  - similar conventions for 2-D/3-D arrays at other nodes (e.g. *array2duw*, *array3dv1*)
    To save memory, these arrays with "dummy names" can be used more than once in a routine with different purposes. For the same reason, the same auxiliary array may be used at different nodes in which case the *c*, *u*, ... is omitted in the name to avoid confusion, e.g. *array2d1*.
- In case a global or local array is defined at different nodes, add the extension *atc*, *atu*, ... at the end, e.g. *depmeanatc*, *depmeanatu*, *depmeanatv .*
- Vector components defined at velocity nodes have a *u, v*, *w* as first character in their name denoting respectively the X-, Y-, Z-component and *atu*, *atv*, *atw* at the end, unless the array is defined at its natural node. For example *uvel*, *vvel*, *wvel* (or *wphys*) are the components of the 3-D current, *ubstresatv* the X-component of the bottom stress (interpolated) at the V-node.
- The following reserved definitions apply for array index variables:
  - *i*, *j*, *k* : X-, Y-, Z- (vertical) index of arrays defined on the model grid
  - *ii, jj* : index of arrays defined at U-, respectively V-open boundaries
  - *icon* : element of a tidal array (e.g. *zetampobu(ii,icon)*)
  - *f* : sediment fraction
  - *iproc* : process number (i.e. element of an array dimension whose size equals *nprocs*)
  - *ivar* : variable index (in case of multi-variate arrays)
- The following (integer) name forms have special meanings and are therefore reserved:
  - *iarr_** : model variable key ids
  - *icon_** : tidal constituent key ids
  - *ics_** : key ids for initial conditions files
  - *ierrno_** : key ids for error codes
  - *igrd_** : key ids for surface grids
  - *io_** : forcing file key ids
  - *iopt_** : model switches
  - *itm_** : timer key ids used for writing a timer report
- In view of parallel applications, some scalars and arrays may have both a local (defined on the process sub-domain) and a global (defined on the full grid) meaning. To avoid confusion the string *loc* or *glb* may be appended at the name of the variable. The string may be omitted in one or both cases (e.g. *nobu* and *nobuloc*) if no confusion is possible. For example, with exception of the bathymetric array *depmeanglb*, all arrays defined on the model grid are local so that only one name (for the local array) needs to be given, e.g. *temp*, *sal*. In other cases, the extension applies only for one definition, e.g. *iobu* for global and *iobuloc* for local.The choice in that case is left to the user. The grid index conventions, described above, become *iloc*, *jloc*, *iiloc*, *jjloc* for local array elements and *iglb*, *jglb*, *iiglb*, *jjgl*b for global array elements.
- It is highly recommended (for transparency) to create some "similarity" in name definitions. For example, since *iobu*, *jobu* are the (global) index coordinates of the open boundary locations and *iobuloc*, *jobuloc* the local equivalents, one may use *ithdu*, *jthdu* as (global) index coordinates of thin dams at U-nodes and *ithduloc*, *jthduloc* for the local equivalents.
- It is recommended to use the same name for arguments which appear in more than one routine with the same purpose, e.g.
  - *cnode* : type of node at which the routine is applied. Note that this type of argument is declared as a string of 3 characters, e.g. "*c* ", "*u* ", "*uv* ".
  - *varid* : key id of a model variable
  - Arguments in the different versions of the same generic routine should have the same or similar names.
  - ...

- **Program routines**
  - Names of program routines (subroutines and functions) are, by default, in lower case letters. Underscores are allowed (and even recommended) to separate different words in the name (*baroclinic_gradient*). Upper case can be used exceptionally for clarity, such as *Carr_at_U* which interpolates a C-node array at the U-nodes.
  - Some consistency in name giving is highly recommended. This applies if a routine has an analogue elsewhere in the code, e.g.

    > *read_vars, write_vars; read_phsics, read_sedics;*
    > *temperature_equation, sediment_equation*
    > *define_2dobc_data, define_surface_data, define_profobc_spec, define_nstgrd_locs*
    > *define_out0d_vals, define_sed0d_vals*

  - Always give useful names explaining the meaning, e.g. *median_particle_diameter .*
  - User-defined routines within a *Usrdef_* file must have a name starting with *usrdef_* .

Declaration of variables

- The first part of the code in a routine consists of the following sections:
    - USE statements
    - IMPLICIT NONE statement (in module routine files this statement is given before the CONTAINS statement and applies therefore for all routines in the file)
    - Type declarations of the arguments (if any)
    - Type declarations of local variables
    - Type declaration statements in COHERENS have the following syntax (note that the ANSI standard allows different forms)

        > type [,att, ...] :: var1 [,var2, ...]

        where

        - type: data type of the variable(s) (e.g. INTEGER)
        - att: one or more attribute(s) of the variable(s):
            - INTENT: must be used in COHERENS for arguments (although not explicitly required by the ANSI standard) , and takes one of the forms: INTENT(IN), INTENT(OUT) or INTENT(INOUT).
            - SAVE: the value of the variable is saved after the routine is exited (not used for arguments). Note that ALLOCATABLE arrays are always declared with the SAVE attribute (see below). The SAVE attribute can be omitted in module declarations, provided that a SAVE statement appears after the last declaration statement.
            - ALLOCATABLE: to declare allocatable arrays(s) (not for arguments)
            - DIMENSION: used to define the array shape(s)
            - PARAMETER: named constant whose value cannot be changed by the program (not for arguments). The attribute is only used for scalars. It is strongly recommended not to use PARAMETER constants for array dimensioning (as done in FORTRAN 77), except for a few arrays whose dimensions are given by constants defined in *syspars.f90*.
            - var1,...: variables sharing the same type and attributes listed in alphabetical order
    - It is remarked that character (string) variables have an additional attribute of the form (LEN=?) where ? equals * or the length of the character string.
    - The KIND attribute is given only for DOUBLE REALs (KIND=long_type), DOUBLE INTEGERs (KIND=longint_type) and COMPLEX (KIND=complx_type) data types.
    - DERIVED TYPE scalars and arrays are declared in the same way with the appropriate TYPE definition.
    - Examples

```
CHARACTER (LEN=12) :: ctype
CHARACTER (LEN=lenname), DIMENSION(MaxProgLevels) :: procname
INTEGER, PARAMETER :: lentime = 21
INTEGER, SAVE :: iunit
REAL :: xtemp, ytemp
REAL, DIMENSION(ncloc,nrloc) :: array2dc1
REAL, SAVE, ALLOCATABLE, DIMENSION(:,:) :: array2dc2, array2dc3
TYPE (VariableAtts), SAVE, ALLOCATABLE, DIMENSION(:) :: varatts
```

Modules and module routines

- **Modules**
    - Modules are places where parameters and arrays are defined which are "global" to the program.
        - Arrays in modules are declared as ALLOCATABLE. It is recommended not to change this practice. The reason is that the array dimensions are defined by the user or defined by the program itself. Moreover, when the program is used in parallel mode, the dimensions of model grid arrays depend on the sizes of each local sub-domain. For example, the dimensions of the global domain *nc, nr*, *nz* are defined in *usrdef_mode_params*, the local horizontal dimensions *ncloc*, *nrloc* (which are actually used for dimensioning of model grid arrays) are defined afterwards by the program. The number of sediment fractions *nf*, used as last dimension for sediment arrays is defined in *usrdef_sed_params*. Exceptions to the rule are a few arrays whose dimensions are defined by system parameters in *syspars.f90* (e.g. the array *modfiles*).
- **Module routines**
    - Main question for a developer is whether a newly created routine should be defined as an external or a module routine. The answer is twofold:
        - Module routines are a kind of "auxiliary" or "library" routines serving general purposes and are called throughout the program. External routines, on the other hand, constitute the actual model (updating of the hydrodynamics, temperature, salinity and concentration fields, turbulence, update of forcing data, ...). The following module routine files are available:
            - *array_interp.f90*: array and scalar interpolation on the model grid
            - *cf90_routines.F90*: library of netCDF routines
            - *check_model.f90*: check model parameters and arrays for errors
            - *check_sediments.f90*: check sediment model parameters and arrays for errors
            - *cif_routines.f90*: generic routines for reading from or writing to a CIF
            - *comms_MPI.F90*: library of MPI routines
            - *datatypes_init.f90*: initialise derived types arrays of different types
            - *default_model.f90*: default settings of (physical) model parameters and arrays

        - *default_sediments.f90*: default settings of sediment model parameters and arrays
        - *error_routines.F90*: routines for checking and writing of model errors
        - *fft_library.f90*: library routines for performing Fast Fourier Transforms (currently not used)
        - *grid_interp.F90*: routines for performing interpolations from and to an external data grid
        - *grid_routines.f90*: various routines for operations on the model grid
        - *inout_paral.f90*: I/O operations for parallel applications
        - *inout_routines.f90*: I/O operations
        - *math_library.F90*: miscellaneous mathematical functions and operations
        - *model_output.f90*: routines for defining standard output (physical) model data
        - *modvars_routines.f90*: attributes of model variables and forcing files
        - *nla_library.F90*: routines for solving linear systems of equations
        - *paral_comms.f90*: routines for parallel communications
        - *paral_utilities.f90*: routines for performing operations (MAX, MIN, SUM) in parallel mode
        - *reset_model.F90*: (re)set model parameters and arrays depending on settings by the user
        - *reset_sediments.f90*: (re)set sediment parameters and arrays depending on settings by the user
        - *rng_library.f90*: routines for generating random numbers
        - *sediment_output.f90*: routines for defining standard output sediment data
        - *sedvars_routines.f90*: attributes of sediment model variables and forcing files
        - *time_routines.f90*: calendar date and time routines
        - *utility_routines.f90*: miscellaneous utility routines
    - Second reason for using module routines is that additional options are available. These options can, according to the ANSI standard, also used for external routines by creating so-called explicit interfaces in the calling program. This method is far from elegant (and prone to errors) and is therefore not used in COHERENS. Note that some compilers allow the use of the above options for external routines without explicit interface. Besides being non compatible with the ANSI standard, unforeseen errors may occur. The following options are then reserved for module routines only:
        - optional arguments
        - argument keywords
        - generic routines and operators (the latter is rarely used in the COHERENS code)
        - assumed-shape arrays

Arrays

- **Initialisation**
    - Array initialisation is important. The reason is that if an array (or array section) appears on the right of an assigment expression, all elements of the array (or array section) must have a defined value by a previous assignment expression with the same array (section) on the left. If this is the case, it may happen that no error occurs on some compilers (but giving useless NANs instead), whereas a segmentation or floating error is issued on other compilers.
    - The arrays, declared in modules, are always initialised with a default value (0.0 for reals, 0 for integers, .FALSE. for logical arrays and '' for string arrays). This initialisation is performed just after allocation. Derive type arrays are initialised using the routines in *datatypes_init.f90*.
    - If an array assignment expression appears within a WHERE construct, only the array elements where the mask is .TRUE., need to be defined *a priori*. The developer may decide whether it is useful to initialise local arrays or not. Important to note is the mask used in the WHERE statement must be defined everywhere !
    - Important to note also is that all elements of arrays or arrays sections need to be defined when written to an output file or used in a MPI send communication.
- **Assignment**
    - To assign values to an array (section) use array assignment statements. The expression on the right must have the same shape (which is more strict than being of the same size) or a scalar. The reason for preferring array assignment in stead of the traditional method using DO loops is that a more optimum use of internal memory may be expected. There are however some restrictions (discussed next).
    - For optimum use and transparency the following rules apply for assignment
        - Lower and upper bounds in array sections are omitted if they coincide with bounds in the array's definition.
        - Bounds and parentheses are omitted if all lower and upper bounds are the same as the ones by which the array is declared (allocated):

```
REAL, DIMENSION(ncloc,nrloc) :: array2dc1
REAL, DIMENSION(0:ncloc,nrloc) :: array2dc2
REAL, DIMENSION(ncloc,0:nrloc) :: array2dc3

array2dc1 = ...; array2dc2 = ...
WHERE (array2dc1.GT.0.0)
   array2dc3(:,1:nrloc) = array2dc2(1:ncloc,:)/array2dc1
END WHERE
```

    - Most assignments are made for model grid arrays. To exclude dry points from the calculation, the assignment must be within a WHERE block. Since the mask in the WHERE statement is a 2-D horizontal array, assignments for 3-D arrays, including a vertical dimension, can only be coded by putting the WHERE block inside a vertical DO loop, e.g.

```
k_2111: DO k=1,nz+1
   WHERE (maskatc_int)
      wadvatw(:,:,k) = wvel(1:ncloc,1:nrloc,k)
   END WHERE
ENDDO k_2111
```

- Masks can be used in addition to avoid division by zero or other floating point exceptions (e.g. square root of negative value(s)).
- WHERE blocks have some disadvantages
    - Only arrays assignments are permitted within the block.
    - No mask can be provided for an ELSEWHERE statement.
    - Nested WHERE blocks are not permitted according to the FORTRAN 90 ANSI standard.
    - Some intrinsic functions are not allowed within a WHERE block (compiler dependent).
    - In cases where no suitable (efficient) WHERE construct is available, explicit DO loops can be used. For example

```
k_161: DO k=1,nz
j_161: DO j=1,nrloc
i_161: DO i=1,ncloc
   IF (ANY(nodeatu(i,j-1:j,k).GT.2).AND.&
     & ANY(nodeatv(i-1:i,j,k).GT.2)) THEN
      IF (COUNT(nodeatu(i,j-1:j,k).LE.1).GT.0.AND.&
        & COUNT(nodeatv(i-1:i,j,k).LE.1).GT.0) THEN
         nodeatuv(i,j,k) = 0
      ELSE
         nodeatuv(i,j,k) = 4
      ENDIF
   ELSEIF (ANY(nodeatv(i-1:i,j,k).GT.2)) THEN
      nodeatuv(i,j,k) = 3
   ELSEIF (ANY(nodeatu(i,j-1:j,k).GT.2)) THEN
      nodeatuv(i,j,k) = 2
   ELSEIF (COUNT(nodeatu(i,j-1:j,k).GT.1).GT.0.AND.&
         & COUNT(nodeatv(i-1:i,j,k).GT.1).GT.0) THEN
      nodeatuv(i,j,k) = 1
   ELSE
      nodeatuv(i,j,k) = 0
   ENDIF
ENDDO i_161
ENDDO j_161
ENDDO k_161
```

As shown above, the order of nested loops is important. Following the FORTRAN rule which states that arrays are stored in memory with the first (leftmost) index as the fastest and last (rightmost) index as the slowest varying index, the inner loop is over the X-index (first dimension), the middle over the Y-index (second dimension) and the outer loop over the vertical index (third and last dimension).

- **Allocation**
    - Arrays with a global scope, declared in modules, are (practically) always declared as ALLOCATABLE. Most arrays are allocated in *allocate_mod_arrays* (and its equivalents *allocate_sed_arrays*, *allocate_bio_arrays*).
    - The following rules apply for local arrays:
        - Except for the cases, mentioned in the next lines, local arrays are declared as ALLOCATABLE or as automatic (with specified dimensions) depending on whether the compiler option -DALLOC is defined or not. These options are implemented in the code via the *#ifdef* ALLOC ... *#endif* /*ALLOC* block statements.
        - Some local arrays must always be ALLOCATABLE: arrays which need to be SAVEd after exiting of the routine and arrays requiring large amounts of memory (such as arrays defined on the global computational grid for parallel applications) .
    - To optimise internal memory, it is recommended to use conditional allocation (by means of an IF statement) in case the array is used only for certain value(s) of one or more switches.
    - Local allocatable arrays must always be declared with the SAVE attribute, even when saving is not required. The reason is to avoid that arrays are stored on the stack and on some machines the stack size is very limited.
    - The following procedure is adopted in COHERENS for allocation. Each line with an ALLOCATE statement is followed by a line for error checking, followed (eventually) by a line for initialisation

```
ALLOCATE(array2dc(ncloc,nrloc),STAT=errstat)
CALL error_alloc('array2dc',2,(/ncloc,nrloc/),real_type)
array2dc = 0.0
```

Note that the lower bound is omitted in the ALLOCATE statement if it equals 1. The STAT=*errstat* keyword argument is needed (in this format) to enable the error checking on the second line.

- All arrays which have been allocated must be deallocated before the end of the program to avoid an error when the program attempts to allocate an array which hasn't been deallocated. The rule applies also for arrays with global scope and arrays which need saving since the current simulation may be followed by a new one, activated by reading of a new title from the *defruns* file.
    - Global arrays are deallocated in *deallocate_mod_arrays* (and its equivalents *deallocate_sed_arrays*, *deallocate_bio_arrays*)
    - Not SAVEd local arrays are deallocated just before exiting the routine.
    - SAVEd local arrays are allocated on first call and deallocated on last call of the routine. In the rase case that it is unknown when (or on which conditions) this last call is made, an exception to the general rule can be made by deallocation just before the allocation, e.g.

      ```
      IF (nt.EQ.0) THEN
         IF (ALLOCATED(array2dc)) DEALLOCATE (array2dc)
         ALLOCATE(array2dc(ncloc,nrloc),STAT=errstat)
         ...
      ENDIF
      ```

      Note that, in case of conditional allocation, i.e. depending on the value of some switch, this deallocation must be made prior to the IF conditional statement, since the value of the switch may have changed in a subsequent run.

    - Except for the special case above, deallocation of arrays, which are allocated conditionally, must be deallocated using the same IF condition.
- Important to note is that no values can be assigned to ALLOCATABLE arrays which are not been allocated, or to (ALLOCATABLE or any other) arrays of size zero. Note that violation of these constraints can lead to nasty bugs which are sometimes very difficult to find !

I/O operations

- **Main code**
    - Never use the FORTRAN OPEN statement to open a file! The reason is that file units are defined internally by the program to avoid that the same unit is used for different files. In case of a file whose attributes stored in a DERIVED TYPE variable of type *FileParams*, use *open_filepars* for opening the file. This includes all forcing files with attributes stored in *modfiles* and the files for standard user output whose attributes are stored in *out0d*, *out2d*, ...Monitoring (and other) files are opened with *open_file.*
    - Files opened with *open_filepars* are closed with *close_filepars*. Files opened with *open_file* are closed with *close_file*.
    - The program uses the standard generic routines *read_vars* and *write_vars* for reading and writing data in standard format. This applies for forcing data stored in files using standard COHERENS format and all user-defined output (except those defined in *usrdef_output*). Note that, in view of parallel applications, these routines are not called directly since a distribute operation has to be made after each read and a combine communication before each write operation (see below). This means that, in practice, *read_vars* is replaced by *read_distribute_mod* and *write_vars* by *combine_write_mod* (or other equivalent routines depending on the type of combine communication).
    - Monitoring data (log file and error file(s), warning and timer report files) are written with the FORTRAN WRITE statement. Note that these files are always in ASCII format.
- ***usrdef_* routines**
    - As before, OPEN and CLOSE calls are not allowed. Both *open_file* and *open_filepars* are permitted. However, it is recommended to use the latter for forcing files. A file opened with *open_file* (*open_filepars*) must be closed with *close_file* (*close_filepars*).
    - FORTRAN READ and WRITE statements are allowed. In case of input files in netCDF format, it is recommended to make calls using the aliases (starting with *cf90_*) defined in *cf90_routines.f90* instead of the NF_90 routines from the netCDF library.
    - *usrdef_** routines used to define time series data (*usrdef_2dobc_data*, *usrdef_profobc_data*, *usrdef_surface_data*) are called two or three at the initial time:
        - On the first call the data file is opened. If the data are defined within the routine itself, the *status* attribute of the corresponding forcing file has to be set to 1. If the data are obtained without external data file, only the *status* attribute has to be set.
        - A first series of data is read on the second initial call.
        - If the data are to be interpolated in time, a second series of data is read on the third call.
    - The previous comments obviously no longer apply for data files, already available in standard COHERENS format, in which case the file data are no longer read from a *usrdef_* routine.
    - Rules for parallel applications are given below.

Parallel code

- **Halos**
    - A general rule in COHERENS is that model grid arrays are only defined "locally" on each sub-domain. When new model grid arrays are introduced in a new development, the developer then needs to know, firstly, whether the array needs a halo and, secondly, what are the required halos sizes. The following rules apply:
        - An array representing a variable which is updated at each time step by solving an advection-diffusion transport equation, must be defined with a halo in all directions of size given by the model parameter constant *nhalo* (which equals 2).
        - Arrays need a halo in specified directions if the array is interpolated somewhere in the code from its natural node to

another node:
        - A C-node array, interpolated at U- or V-nodes, needs a halo of size 1 at its western or southern node.
        - A U- or V-node array, interpolated at the C-node, needs a halo of size 1 at its eastern or northern boundary.
        - Similar rules apply for other types of interpolations.
    - Other, more exceptional, cases, requiring halos are applications of open boundary conditions, horizontal averaging (e.g. as part of a nesting precedure).
    - Note that not all arrays need a halo.
  - It is noted that in the current version of COHERENS a halo of size *nhalo* is sufficient for all purposes.
  - In the current version, the size of a halo cannot be larger than 2 in each direction. If a larger size is required as part of a new development (e.g. a higher order advection scheme), the developer must be aware that this will require a complete revision of all communication routines in the program ! This means, in particular, that the value of the parameter *nhalo* cannot be changed without concertation with MUMM.
  - The golden rule for developers is that arrays with a halo are only defined directly at points inside the local computational domain, e.g. by assignment or reading from a data file. The array sections within the halo and therefore outside the local domain are obtained by making exchange/distribution communications with the neighbouring processes (see below).
- **Exchange communications**
  - Exchange communications are the core business in the parrallel version of COHERENS. Exchange calls are required to update an arrays within its halo. This is performed by a call to the generic routine *exchange_mod*. This call must be made as soon as the array has been (re)defined inside its local domain to ensure that all sections of the array are updated at the same (time) level.
  - It is highly recommended to update array halos by exchange communications only, even when it seems easy to make direct updates without communications. If, nevertheless, the developer wants to make an exception, he/she must be sure that all arrays used in assignment expressions are updated at the same level and have at least the same halo sizes as the array being updated.
  - Unless a development task consists of making changes to the schemes for horizontal advection and diffusion, time integration of a transported variables, the COHERENS communication system and the routines for writing output, the only communication calls which need to be implemented, are exchange calls at the end of routines for those array(s) whose internal values have been updated.
- **Other communications**
  - Distribute operations are only (except for one particular case in *update_surface_data*) used for the distribution of initial conditions. Since the data are read by all processes, this involves no real communications.
  - Combine communications are used for the construction of global arrays from its local parts. This occurs (only) when data (user defined or for nesting) are written to an output file by the master process.
  - Copy communications are implemented in the communication library, but currently not used in COHERENS.
- **Local versus global**
  - Besides data defined on the whole model grid, the model setup may also require definitions of specific locations with a special purpose (open boundaries, thin dams, discharge locations, ...). These locations are defined globally, i.e. stored into global arrays. The program then needs to convert these arrays (and their sizes) into the local equivalents. The example below explains the procedure in case of weirs at U-nodes:

```
!---allocate
1 :ALLOCATE (ind(numwbaru),STAT=errstat)
2 :CALL error_alloc('ind',1,(/numwbaru/),int_type)
3 :ALLOCATE (jnd(numwbaru),STAT=errstat)
4 :CALL error_alloc('jnd',1,(/numwbaru/),int_type)
5 :ALLOCATE (indexarr(numwbaru),STAT=errstat)
6 :CALL error_alloc('indexarr',1,(/numwbaru/),int_type)

!---define local arrays
7 :iproc_110: DO iproc=1,nprocs
8 :   l = 0
9 :   ii_111: DO ii=1,numwbaru
10:      i = iwbaru(ii); j = jwbaru(ii)
11:      IF (local_proc(i,j,iproc)) THEN
12:         l = l + 1
13:         ind(l) = i-nc1procs(iproc)+1
14:         jnd(l) = j-nr1procs(iproc)+1
15:         indexarr(l) = ii
16:      ENDIF
17:   ENDDO ii_111
18:   IF (idloc.EQ.idprocs(iproc)) THEN
19:      numwbaruloc = l
20:      ALLOCATE(iwbaruloc(numwbaruloc),STAT=errstat)
21:      CALL error_alloc('iwbaruloc',1,(/numwbaruloc/),int_type)
22:      ALLOCATE(jwbaruloc(numwbaruloc),STAT=errstat)
23:      CALL error_alloc('jwbaruloc',1,(/numwbaruloc/),int_type)
24:      ALLOCATE(indexwbaru(numwbaruloc),STAT=errstat)
25:      CALL error_alloc('indexwbaru',1,(/numwbaruloc/),int_type)
26:      IF (numwbaruloc.GT.0) THEN
27:         iwbaruloc = ind(1:numwbaruloc)
28:         jwbaruloc = jnd(1:numwbaruloc)
29:         indexwbaru = indexarr(1:numwbaruloc)
30:      ENDIF
31:   ENDIF
32:ENDDO iproc_110

!---deallocate
33:DEALLOCATE (ind,jnd,indexarr)
```

- Since the size of the local arrays are unknown a priori, local results are stored in temporary arrays.
- Local arrays are allocated with the global dimension (lines 1-6), since the local one is still unknown.
- Local positions are stored in the temporary arrays on lines 9-17 where
  - *iwbaru*, *jwbaru* are the arrays, defined by the setup, containing the global index positions
  - *local_proc* returns .TRUE. if the grid point with global indices (i,j) is located within the local sub-domain (excluding halo) with process number *iproc.*
- Once the local size *numwbaruloc* is known, the local arrays can be allocated (lines 18-30). Note that the mask in the IF statement on line 18 only evaluates to .TRUE. on one sub-domain.
- Values are copied from the temporary arrays into the ones (with the correct size) which will be used throughout the code.
- Temporary arrays are deallocated, once they become redundant (line 33).
- The array *indexwbaru* is a so-called index mapping array which maps a local position index into a global one. These types of arrays are of importance in the program because they allow to use global arrays throughout the code and avoid the need to define a local equivalent for each global array. For example, open boundary data are defined globally, as illustrated below where all arrays are global except for the locally defined arrays *iobuloc*, *jobuloc* and *indexobu* maps the local open boundary index *iiloc* into the global one.

```
iiloc_110: DO iiloc=1,nobuloc
   i = iobuloc(iiloc); j = jobuloc(iiloc)
   ii = indexobu(iiloc); ic = MERGE(i,i-1,westobu(ii))
   isign = MERGE(1,-1,westobu(ii))
   iu1 = MERGE(i+1,i-1,westobu(ii))

!  ---define parameters
   SELECT CASE (ityp2dobu(ii))
      CASE (3,8:13)
         IF (iloczobu(ii).EQ.1) THEN
            is = 2
         ELSEIF (iloczobu(ii).EQ.2) THEN
            is = 1
         ENDIF
   END SELECT
```

Other examples can be found in the code.

- Forcing data, like surface data, are defined globally by the setup and then spatially interpolated on the local sub-domain which is done by the program, once the locations of the external data grid is known to the program.
- Important to note that all arrays, parameters, input forcing data, defined as part of model setup are defined globally (either through CIF or via a *usrdef_* routine). Only, unfortunate, exception are the initial conditions which need to be defined in *usrdef_phsics* (and its equivalents *usrdef_sedics*, *usrdef_bioics*) on a local basis.

Model setup

- The first question for a developer is whether the new development can be constructed as a separate outside compartment coupled with the existing code by interfaces. Examples are a sediment transport model, microbiological model, particle tracer model, ice and wave models, ...
  - New compartments are implemented into the existing code by creating a new directory, say *new_mod* (e.g. *sediments*) within the *code* directory in the COHERENS file system. Note that, since this has important consequences for the svn versioning system, such changes to the file system are only allowed in concertation with MUMM !
  - The source code of the new compartment is located in a new source directory, *new_mod/source*.
  - Files for compilation are located in a new directory *new_mod/comps* .
  - Coupling with the main COHERENS code is achieved by insertion of a number of general (interface) routine calls.
  - Setup parameters are defined in a separate CIF or *usrdef*___**params *routine (as part of a new* Usrde**f{_}**{}**_*) file.
  - The principle of the above procedures is to allow coupling of COHERENS with different kinds of external models for sediments, biology, ... by other developers.
- If the new development can be integrated within the existing file system, the next question is whether the setup can de defined in the existing *usrdef_** routines or requires the use of additional *usrdef_** routines. The following guidelines can be given
  - If the new implementation requires (only) the addition of few additional (scalar) parameters, these can be defined in *usrdef_mod_params* (or model CIF).
  - If the new implementation requires the definition of a series of setup arrays (e.g. discharge locations) or new types of forcing data, new *usrdef_** routine(s) must be implemented, located in a new *Usrdef_** file. The name of the routine(s) and file must start with the prefix *usrdef_*, respectively *Usrdef_* and the suffix must clearly refer to the purpose.
  - Note that surface forcing data must always be defined in *usrdef_surface_data*. If a new type of surface forcing is implemented, the developer must define a new key id (*io_*) f*or the surface forcing and key id for the surface grid (*igrd_*)* in *iopars.f90*.
  - In the same way, (3-D) open boundary forcing must always be defined in *usrdef_profobc_spec* and *usrdef_profobc_data*. If needed, a new forcing key id added in *iopars.f90*.
  - Arrays used for setup are always declared as ALLOCATABLE. To allow allocation, before the *usrdef_*, routine, where they are defined, is called, the array dimensions must be defined as parameters in *usrdef_mod_params* or its equivalents *(usrdef_sed_params*, *usrdef_bio_params*, ..., in case of coupling with an external model).

Miscellaneous

Besides the rules and guidelines given above, this section provides additional guidelines and recommendations to improve the efficiency, optimisation, transparancy (readability), portability of the code and to prevent errors in the code *a priori*.

- Always use the ANSI FORTRAN 90 standard. FORTRAN 95 extensions such as FORALL loops and nested WHERE statements, are strictly forbidden.
- INTEGER and REAL variables are declared without KIND specification (except in a very few cases where double precision is required). On most machines this implies a 32-bits representation. The user is free to increase accuracy to 64-bits by adding the appropriate compiler options in *compilers.cmp*. Important to note is that the communications and I/O routines in COHERENS are generic and (currently) only support the single precision (32-bit) case.
- Use existing (module) routines. There several routines in COHERENS written in a general fashion so that they can be applied for many new implementations:
  - Good examples are the (scalar) transport routines in *Transport_Equations.F90*. These routines solve transport equations of the advection-diffusion type with source terms (and possible sink terms in the future) supplied by the user and other arguments for

    open boundary conditions, ...In this way they can be generally applied. Further developments involving changes in the numerical procedures (implicit discretisations, time integration, advection schemes, ...) will be implemented through these routines.
  - COHERENS includes various utility (module) routines for I/O, parallel communication, grid interpolation, ...Some of these routines have optional arguments to cover many kinds of applications (even for the future). Moreover, they can (by now) be considered as (almost) completely debugged.
- Avoid to use array segments as arguments in routine calls where possible. Array segments are passed by value, i.e. a copy of the segment is created in memory on input and the result is (in case of an INTENT(OUT) argument) copied again to the actual argument. Full arrays are passed by reference (i.e. directly through memory) which is much more efficient.
- Don't use the routines of the netCDF and MPI libraries directly but use the aliases, defined in *cf90_routines.F90* and *comms_MPI.F90*. The reason is to avoid major changes at different places in the code, each time a new library version is implemented.
- It is advised not to make modifications to the existing routines, unless it is really necessary and has no impact on other parts of the code. In case these changes have a reasonable impact, concertation with MUMM (and other partners) is required. Of course, these restrictions do not apply when these routines are changed as part of a development task.
- Try to reduce the number of arithmetic operations (as discussed e.g. below).
- Important aspect of an optimised code is that repeated calculations must be avoided even at the expense of clarity. Some examples are given below.
  - Array assignments or array expressions within a loop which do not depend on the loop index should be moved outside the loop.

```
WHERE (maskatc_int)
   array2dc=depmeanatc(1:ncloc,1:nrloc)/deptotatc(1:ncloc,1:nrloc)
END WHERE
k_611: DO k=2,nz
   WHERE (maskatc_int)
      array3dw(:,:,k)=(gscoordatw(1:ncloc,1:nrloc,k)-array2dc)*&
                     & wvel(1:ncloc,1:nrloc,k)
   END WHERE
ENDDO k_611
```

or

```
k_511: DO k=1,nz+1
   WHERE (maskatc)
      array3dw(:,:,k) = MAX((1.0-ctotatw(:,:,k))**n_RichZaki,0.0)
   END WHERE
ENDDO k_511

f_512: DO f=1,nf
k_512: DO k=1,nz+1
   WHERE (maskatc)
      wfall(:,:,k,f) = array3dw(:,:,k)*wfall(:,:,k,f)
    END WHERE
ENDDO k_512
ENDDO f_512
```

Note that the same mask must apply inside and outside the loop.

  - Mask expressions in WHERE statements which are used more than once should be stored in an auxiliary array

```
maskatu  = node2du(1:ncloc,1:nrloc).GT.1
...
WHERE (maskatu)
   ...
END WHERE
...
WHERE (maskatu)
   ...
END WHERE
```

  - Substitute the result of numerical expressions involving only scalars in an array expression, or store the result into a scalar variable before evaluating the array expression.

```
xpi = 0.085*pi
f_320: DO f = 1,nf
   WHERE (maskatu)
      peng = excesshearatu(:,:,f)/&
          & (excesshearatu(:,:,f)**4+
          &(xpi*taunormatu(:,:,f))**4)**0.25
      ...
   END WHERE
ENDDO f_320
```

  - Reduce expressions where possible even at the expense of transparancy, e.g.

    ```
    delta = awave*(awave/ks)**(-0.25)
    ```

    should be replaced by

    ```
    delta = awave**0.75*ks**0.25
    ```

- Avoid divisions by zero, e.g.

  ```
  A/(1+A/B)
  ```

  should be written as

  ```
  A*B/(B+A)
  ```

- Floating point comparison are never exact and must be done with a tolerance

  ```
  eps = 2.0*SPACING(depmean_flag)
  seapoint = MERGE(.TRUE.,.FALSE.,&
           & ABS(depmeanatc(0:ncloc+1,0:nrloc+1)-depmean_flag).GT.eps)
  nodeatc(1:ncloc,1:nrloc) = MERGE(1,0,seapoint(1:ncloc,1:nrloc))
  ```

- Writing floating data in ASCII format can lead to errors due to truncation. For example, if -99.9 represents a data flag for the bathymetry, the value appearing in the data file may be something like -99.90002 which is not exactly the same. This can be resolved by defining a tolerance as described above.
- Do not use array elements as array index, e.g. replace

  ```
  xvar = depmeanatu(iobu(ii),jobu(ii))
  ```

  by

  ```
  i = iobu(ii); j = jobu(ii)
  xvar = depmeanatu(i,j)
  ```

- For an optimal code, do not make routine calls within DO loops, unless there is no alternative. Notable exceptions in the code are the *usrdef_** routines, in e.g. *time_series*, used for defining output at specific output locations.
- For transparancy, DO loops are named. The name has a standard format derived from the section number.

```
!
!1.7 Source terms
!----------------
!

k_170:DO k=klo,kup
   WHERE (maskatc_int)
      source(:,:,k) = delt3d*source(:,:,k)
   END WHERE
ENDDO k_170
```

In case of more than one loop within the same section use consecutive numbering based on the section number.

```
!4.1 X-derivative terms
!----------------------
!
!---time derivative, corrector term
k_411: DO k=klo,kup
   WHERE (maskatc_int)
      tridcfc(:,:,k,2) = 1.0
      tridcfc(:,:,k,4) = psic(1:ncloc,1:nrloc,k) + ucorr(:,:,k)
   END WHERE
ENDDO k_411

...

!---update psic_A
k_412: DO k=klo,kup
   WHERE (maskatc_int)
      psic_A(1:ncloc,1:nrloc,k) = tridcfc(:,:,k,4)/tridcfc(:,:,k,2)
   END WHERE
ENDDO k_412
```

- IF blocks with multiple ELSEIF statements should, where possible, be replaced by more transparent SELECT CASE constructs.
- If new types of forcing data need to be defined by a new implementation (discharge locations and data, surface wave input, ...), the attributes must be stored in the existing DERIVED type array *modfiles*. New forcing key ids (of the form *io_**) need then to be defined in *iopars.f90* and the value of *MaxIOTYPES* needs to increased in *syspars.f90*. In this way a series of default settings, checking and other operations are automatically performed by the program. Note that the attributes of forcing files are defined in *usrdef_mode_params* or in the model CIF.
- For clarity, routines in a file are sorted alphabetically.
- Be aware of truncation (rounding) errors.
  - Substracting two positive variables of the same sign or adding two variables of opposite sign always gives a loss of accuracy if the two variables are real and of the same magnitude.
  - Truncation occurs when (tidal) phases are updated by adding frequency times the time step and the time step is much smaller than the period. In the code, this is solved by making the addition in double precision (see routine *update_2dobc_data*).
- Parameters and arrays in a CIF are sorted firstly in sections and secondly alphabetically in each section. Each section corresponds to a certain type of purpose of variables. For a model CIF, these sections are model switches, date/time parameters, real and integer model parameters, ...
- Each call to *log_timer_in* in the beginning of a routine must be followed by a call to *log_timer_out* at the end of the routine. Otherwise, the tracing level in the log file becomes erroneous and the program may even crash. Note that, if a timer argument is applied with these routines, the time measured by the program and stored in the corresponding time counter, is given by the time difference between the calls to *log_timer_out* and *log_timer_in* and includes all calls made within the routine.
- Avoid, where possible FORTRAN statements, such as GOTO, CYCLE, EXIT, which redirect the program to other parts of the code.
- A GOTO statement is allowed if error coding is provided from within a routine (instead of using one of the routines in *error_routines.f90*). In that case, the error code must be put after the RETURN and before the END ... statement.

Common bugs

- Syntax errors. These are usually detected by the compiler.
- Typing errors in the name of variables. They can mostly be detected by the compiler since the program does not use implicit typing. Rare exception is that a wrongly typed variable has given the name of another variable (either local or from a USEd module).
- A very common error is the violation of array bounds. It is therefore strongly recommended to compile the program with a debugging compiler option for checking array bounds (usually this is the *check bounds* option). Care must be taken, since the debugger may not detect all violations.
- Floating point exceptions, i.e. division by zero, square root of a negative value, ... This may be detected by setting the appropriate debugging option for floating exceptions and overflow.
- Floating point exceptions, generated by using undefined values in array expressions. Unfortunately, although compilers can detect these

errors, the issued error message is in most cases not very meaningful. In case the option for floating exceptions is not set, the program just produces NaNs. To my knowledge, the only compiler which can produce the exact location of the error is the DIGITAL FORTRAN compiler.

- Assign values to zero SIZEd arrays. These errors produce memory faults which are very nasty because they not produce an immediate error. In stead the program crashes somewhere else in the code with the wrong error message ! When the source of this error is detected, it can be remedied by putting the assignment within an IF block or by replacing the assignment by an explicit DO loop. Note that in parallel mode, arrays may be of size zero on some and size non-zero on other processors.
- A non-allocated array cannot be deallocated whereas an allocated array must be deallocated before it can be allocated again. These errors are usually found by the compiler.
- The type, size or rank of the actual array arguments in a calling routine must match the ones declared for the corresponding dummy arguments in the called routine. In case of generic routine calls, the usual error message is that no routine matching is found. Note that actual arguments of type CHARACTER with the INTENT(IN) attribute, declared with a fixed length in the called routine, must exactly have the same length (add spaces if necessary). Note also that a scalar argument is not the same as a vector or array of SIZE one in the ANSI standard.
- The mask expression must be defined for all its elements and not only at sea points. On the other hand, an array evaluated by a WHERE (without ELSEWHERE) statements becomes only conditionally defined. Floating point exceptions must be prevented, which is not always trivial as shown in this example

```
1 :WHERE (maskatu)
2 :!--convert current bstres from total stress to skin friction
3 :   taucur = bstresatu*(LOG(zbot/zroughatu_sed)/&
                        & LOG(zbot/(0.15*d50atu)))**2
4 :!--convert wave total bstress to skin friction (z0~d50)
5 :   tauwav =  bstresatu_wav*(zroughatu_sed/d50atu)**(-0.52)
6 :   array2d1 = taucur + tauwav
7 :ELSEWHERE
8 :   array2d1 = 0.0
9 :END WHERE

!---phase averaged shear stress
10:WHERE (array2d1.GT.0.0)
11:   taumean = taucur*(1.0+1.2*(tauwav/array2d1)**3.2)
12:ELSEWHERE
13:   taumean = 0.0; taumax = 0.0
14:END WHERE

!---maximum shear stress
15:CALL Carr_at_U(waveuvel(:,1:nrloc),array2d1,1,3,(/0,1,1/),&
16:            & (/ncloc,nrloc,1/),1,iarr_waveuvel,.TRUE.)
17:CALL Carr_at_U(wavevvel(:,1:nrloc),array2d2,1,3,(/0,1,1/),&
18:            & (/ncloc,nrloc,1/),1,iarr_wavevvel,.TRUE.)
19:CALL complex_polar(array2d1,array2d2,xpha=phiwav,maskvals=maskatu,&
20:                 & outflag=0.0)
21:CALL complex_polar(ubstresatu_sed,vbstresatu_sed,xpha=phicur,&
22:                 & maskvals=maskatu,outflag=0.0)
23:WHERE (maskatu)
24:   phi = phiwav - phicur
25:   taumax = SQRT(taumean**2+tauwav**2+&
               & 2.0*taumean*tauwav*COS(phi))
26:ELSEWHERE
27:   taumax = 0.0
28:END WHERE

!---bed load per fraction
29:f_3511: DO f=1,nf
30:   WHERE (taumean.GT.0.0.AND.taumax.GT.taucritatu(:,:,f))
31:      array2d1 = SQRT(taumean)*(taumean-taucritatu(:,:,f))
32:      array2d2 = (0.95+0.19*COS(2.0*phi))*SQRT(tauwav)*taumean
33:      bphix = 12.0*MAX(array2d1,array2d2)
34:      bphiy = 2.28*taumean*tauwav*tauwav*SIN(2.0*phi)&
35:            & /(tauwav**1.5+1.5*taumean**1.5)
36:   ELSEWHERE
37:      bphix = 0.0; bphiy = 0.0
38:   END WHERE
39:ENDDO f_3511
```

- The array array2d1 must be defined everywhere (using ELSEWHERE) before it can be used in a WHERE expression on line 10.
- *taumean* and *taumax* must be defined everywhere since they are used in the WHERE mask on line 30.
- *array2d1* is redefined by the grid interpolation call on lines 15-16. This poses now problem since the grid interpolation routines assign values for all array elements (flags on land points).

    - All arrays appearing in assignment expressions in the WHERE block on lines 30-38 are now properly defined. The array *taucritatu* was already defined (not shown). The array *phi* is defined on line 24 with the mask *maskatu* which is sufficient since the WHERE mask on line 30 is more restrictive than the one on line 23, so there is, again, no problem.
    - *bphix*, *bphiy* need to be defined everywhere since they are used (not shown) using a less restrictive mask.
- To avoid divisions by zero, be sure to use the correct values for the *intsrce* and *intdest* arguments in the routines for performing interpolation on the model grid (file *array_interp.f90*).
- MERGE statements are very practical since they can be used as shortcuts for WHERE blocks. They can also be dangerous since the evaluation is compiler dependent. For example

```
result = MERGE(0.0,A/B,B.EQ.0.0)
```

    - On INTEL compilers the evaluation is robust since the mask *B.EQ.0* is evaluated first and *A/B* is only evaluated and returned in *result* only when *B* is non-zero.
    - On a gfortran compiler, *A/B* is evaluated before the mask yielding a floating exception (division by zero).
    - The following example is taken from the code to see how this kind of problem can be avoided

```
WHERE (xreal.EQ.0.0.OR.ximag.EQ.0.0)
   xmag = MERGE(ABS(xreal),ABS(ximag),ximag.EQ.0.0)
ELSEWHERE
   xmag = MERGE(ABS(xreal)*SQRT(1.0+(ximag/xreal)**2),&
              & ABS(ximag)*SQRT(1.0+(xreal/ximag)**2),&
              & ABS(xreal).GT.ABS(ximag))
END WHERE
```

- Typical errors in the parallel implementation are:
    - Use of local data in the global sense or global data in the local sense. For example,
        - Global initial conditions, obtained by reading from a data file but not distributed on the local grid.
        - Arrays may be of size non-zero on one process domain, but of size zero on another, so that care must taken in assignment expressions. This can be resolved by using explicit DO loops or putting the assignment in an IF block.
    - Arrays defined with improper halo sizes.
    - Omitted exchange communications.
    - Exchange communications with incorrect values for exchange directions and sizes. If the halo size in a direction a larger than the one given in the array's declaration, the array bounds are violated. If the exchange size is lower than in the declaration, some points in the halo are not updated which may (or may not) be erroneous.
    - Read or write operations without the necessary distribute or combine communication.
    - Violation of array bounds when array sections need to be used instead of full arrays in assignments expressions or as arguments in routine calls. In the latter case, care should be taken that the array dimensions of the actual argument comply with those of the corresponding dummy argument.

Testing a new implementation

- The following procedure is recommended for testing new code:
    - Compile the code without defining an application, i.e. using the script *install_test* without -t argument. In this way, rudimentaty bugs, like syntax errors, are easily removed.
    - Run a few of the existing test cases, to verify that the implementation has no side effects on the already existing code.
    - Define new test cases for testing of the new implementations:
        - If the new code consists of several independent implementations, selected by e.g. new model switches, it is recommended to define a specific test case (or test case experiment via the *defruns* file) for each option provided by the new code.
        - It is also advised to design at least one general test case, to test e.g. the combination of different sub-implementations and other functionalities in the program (e.g. inundation schemes).
- Test case parameters must be defined for each new test case.
    - The common procedure in COHERENS is to make these definitions in *usrdef_output*.
    - The results are written to an output file whose name is the name of the experiment (as defined in *defruns*) followed by the suffix *.tst*.
    - These test case parameters should be designed for debugging of the new implementation and all future versions of COHERENS.
    - The aim of the parameters is, besides debugging, to verify the sensitivity of model parameters and switches or to compare with analytical solutions. They should be also sensitive to changes in the code and particular aspects of the new development.
    - The files for setting up the test case (*defruns* and *usrdef_** files) must be located in a subdirectory of the setups directory. The name of the subdirectory should be the same as the name of the new test case as given in *defruns*. The name should preferentially contain no more than 8 characters and refer to the aim of the test.
- All test simulations must be performed with (at least) the following debugging options for the compiler:
    - checking for violation of array bounds
    - checking for floating exceptions and floating overflow.
- Disadvantage of these debugging options is a significant increase of CPU time. To save time, one may launch the test case, firstly, with debugging on, stop the program after a short simulation (say after a few time steps), recompile without debugging and run the test case again until the end.
- Once all test cases are running successfully, the test should be repeated in parallel mode. It is advised to select a few test cases and to run each test more than once on the same machine, each time with a different number of processors. The results must always be exactly

the same.

Documentation

- All documentation is written in LateX code.
- When a new development has been added or the code has been changed and these changes need to be documented for the user, these new developments or other modifications need to be inserted in the existing code at the appropriate places. New chapters only need to be created in case the main code is coupled with a large new module, such as sediments, biology, wave coupling, .. In most cases, however, the newly added features can be documented within the existing chapters:
    - Part II: model description with separate chapters for general physical description, numerical methods, sediments, biology, ...
    - Part III: description of the model code (see manual for details)
    - Part IV: user manual: each chapter covers a specific topic corresponding to a corresponding *Usrdef_* file or *usrdef_* routine(s) or CIF. It is recommended to keep the same formates (style) in the text.
    - Part V: each new test case must documented. In most cases a new chapter needs to be created. Further details are given below.
    - Part VI: reference manual. Each new routine or module is documented in the same format (as found in the LateX files). There are separate chapters for external routines, module routines, *usrdef_* routines and modules with declarations only.
- Formulas in the documentation must exactly match the expressions in the code.
- The LateX documentation is compiled using *pdflatex* (instead of the previously common *latex* command). The advantage is that a pdf file is automatically generated.
- *pdflatex* allows the use of most common formats for figures (*pdf*, *jpg*, *png*, ...). Postscript is, however, not allowed. Encapsulated postsrcript is, in principle, allowed but not recommended since *pdflatex* has to convert the figure to *.eps* format first. This conversion can more easily made by the developer before insertion into the document.
- The following rules are adopted for documenting test cases:
    - A (short) description of the test, allowing the user to understand the aim of the test
    - The documented test case must exactly match the one given in the model code (i.e. using the same parameters, grid resolution, ...)
    - A table with a description of the different experiments (as listed in the file *defruns*) performed with the same test case.
    - A table describing the meaning of each test case parameter.
    - Figures which can easily be reproduced by the user for testing the implementation.
    - A short description of the most important results.
- Bibliographic references are included using the BibTeX utility for LateX documentation. All references need to be defined in the file *references.bib* using the BibTeX format.