Modelling Tools Manual

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
526	2016-04-12		SB

Contents

Abstract

This document describes the modelling tools and approaches used for the new individual-based and agent-based model environment (framework). First, it documents the modelling **modules** that are developed to make coding such models simpler. Second, it outlines the general coding style and an aproach based on **object-oriented** programming.

Modern **Fortran** can be considered as an almost ideal language for agent-based modelling. It is high-level (e.g. it allows to work with whole arrays and slices) and partly object-oriented. It also contains many similar constructions with Python, so the later can be used for rapid prototyping. Nonetheless, it is compiled and strictly typed which makes coding big and complex projects safer. Compilers are easily available, including free GNU gfortran. Recent compilers generate highly efficient and extremely fast machine code. Modern Fortran includes some built-in parallel calculation instructions, and libraries and tools for high performance parallel computations are readily available. As such, Fortran is one of the favourite languages for computation-intensive works.

Document date: 2016-04-12.

SVN address of this document source is:
$HeadURL: https://svn.uib.no/aha-fortran/branches/budaev/HEDTOOLS/BASE_UTILS.adoc $

1 Software tools and requirements

Most tools needed for the model are already available on Linux (e.g. gfortran, make, Subversion, console, midnight commander etc) and are trivial to install using the standard package manager (e.g. `apt-get install gfortran` on Ubuntu). On Windows they can be installed manually from their official web sites. On Mac use homebrew to install many of the utils. Below are some details on the Windows software.

Fortran Compiler (Mandatory)

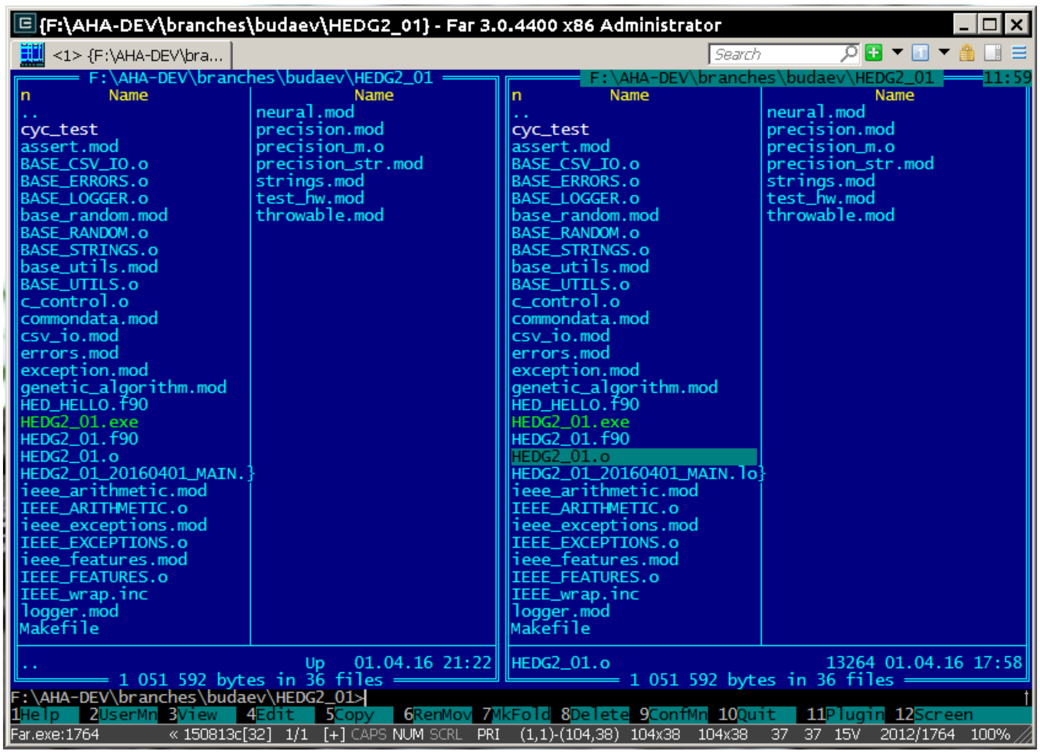
Intel Fortran compiler, a commercial software available at UiB. Intel Fortran is also installed on the UiB HPC cluster fimm. Free **GNU Fortran** distribution along with make and other tools is available from the Equation solution http://www.equation.com/-servlet/equation.cmd?fa=fortran. There is also **Oracle Solaris Studio** combining Fortran compiler and an NetBeans-based IDE, freely available from http://www.oracle.com/technetwork/server-storage/solarisstudio, Linux and Solaris OSs only (no Windows or Mac).

Subversion (Mandatory)

Windows GUI for Subversion is **TortoiseSVN** (supported by UiB IT): https://tortoisesvn.net/. It is very helpful to have also console subversion client software (Tortiuse includes console tools but does not install by default). A good one is SilkSVN https://sliksvn.com/download/.

Console terminal (Highly recommended)

The Windows console (`cmd`) is extremely weak. **Conemu** https://conemu.github.io/ is a much better alternative, especially with the **Far manager**, a two-panel console file manager similar to the ancient Norton commander for DOS (or Midnight commander on Linux): http://www.farmanager.com/download.php?l=en.



It is also very helpful to have (on the Microsoft Windows) the GNU core utilities (`grep`, `cut`, `sed`, `awk` etc.). Some of them are used in the GNU make build system, and some are included with the Equation solutions gfortran (**TODO: have to check what is really necessary**). There are several distributions available, e.g. GnuWin32, Cygwin, MinGW, UnxUtils, Gow, winbash.

Geany (Recommended)

Lightweight IDE, Editor for code and any text files. Works on Linux, Windows and Mac. http://www.geany.org/ Also need plugins: http://plugins.geany.org/

Code::Blocks for Fortran (Recommended)

IDE for Fortran. Works with many compilers, including Intel and GNU gfortran. http://cbfortran.sourceforge.net/. Installation by unpacking into some directory.

Follow: A logfile reading program (Recommended)

Following a logfile while executing a program is done trivially on Linux: `tail -f some_log_file.txt`. There is a Java GUI program for reading log files that works on all major platforms installs by just placing in some directory: Follow. Available from http://sourceforge.net/projects/follow/.

2 Introduction the Fortran modules

Module is just a piece of Fortran program that contains variable or constant declarations and functions and subroutines. Modules are defined in such a simple way:

```
module SOME_MODULE

character (len=*), private, parameter :: text_string = "its value"
integer :: some_variable
real, dimension(:)

contains                       ! subroutines and functions go after "contains"

  subroutine SOME_SUBROUTINE(parameters)
  ...
  end subroutine SOME_SUBROUTINE

end module SOME_MODULE
```

To use any variable/constant/subroutine/function from the module, the program must include the use `MODULE_NAME` statement:

```
use SOME_MODULE
....
```

The AHA modelling tools include several separate modules:

- `BASE_UTILS`
- `CSV_IO`
- `BASE_RANDOM`
- `BASE_STRINGS`
- `LOGGER`
- Error trapping modules
- IEEE Arithmetics modules

BASE_UTILS contains a few utility functions. CSV_IO is for output of numerical data into the CSV (comma separated values) format files. CSV is good because it is human-readable but can still be easily imported into spreadsheets and stats packages (R reads CSV). It also has little file size overhead which is good if huge amounts of data are generated by the model.

Invoking the modules requires the `use` keyword in Fortran. `use` should normally be the first statements before `implicit none`:

```
program TEST

  use BASE_UTILS   ! Invoke the modules
  use CSV_IO       ! into this program
```

```
  implicit none

  character (len=255) :: REC
  integer :: i
  real, dimension(6) :: RARR = [0.1,0.2,0.3,0.4,0.5,0.6]
  character (len=4), dimension(6) :: STARR=["a1","a2","a3","a4","a5","a6"]

.........

end program TEST
```

Building the program with these modules using the command line is normally a two-step process:

build the modules, e.g.

```
gfortran -g -c ../BASE_CSV_IO.f90 ../BASE_UTILS.f90
```

This step should only be done if the source code of the modules change, i.e. quite rarely.

build the program (e.g. TEST.f90) with these modules

```
gfortran -g -o TEST.exe TEST.f90 ../BASE_UTILS.f90 ../BASE_CSV_IO.f90
```

or for a generic F95 compiler:

```
f95 -g -c ../BASE_CSV_IO.f90 ../BASE_UTILS.f90
f95 -g -o TEST.exe TEST.f90 ../BASE_UTILS.f90 ../BASE_CSV_IO.f90
```

A static library of the modules could also be built, so the other more changeable code can be just linked with the library.

Note
The examples above assume that the module code is located in the upper-level directory, so ../ , also the build script or Makefile should normally care about all this automatically.

3 Module: BASE_UTILS

This module contains a few utility functions and subroutines. So far there are two useful things here: **STDOUT**, **STDERR**, **TOSTR**, **CLEANUP**, and **RANDOM_SEED_INIT**.

3.1 Function: TOSTR

TOSTR converts everything to a string. Accepts any numeric or non-numeric type, including integer and real (kind 4 and 8), logical and strings. Also accepts arrays of these numeric types. Outputs just the string representation of the number. Aliases: **STR** (same as **TOSTR**), **NUMTOSTR** (accepts only numeric input parameter, not logical or string)

3.1.1 Examples:

Integer:

```
STRING = TOSTR(12)
produces  "12"
```

Single precision real (type 4)[1]

[1] Note that float point calculations, especially single precision (real type 4) may introduce a rounding error

```
print *, ">>", TOSTR(3.1415926), "<<"
produces >>3.14159250<<
```

Double precision real (type 8)

```
print *, ">>", TOSTR(3.1415926_8), "<<"
produces >>3.1415926000000001<<
```

TOSTR also converts logical type to the "TRUE" or "FALSE" strings and can also accept character string as input. In the latest case it just output the input.

Optional parameters

TOSTR can also accept standard Fortran format string as the second optional **string** parameter, for example:

```
print *, ">>", TOSTR(3.1415926,"(f4.2)"), "<<"
produces >>3.14<<
```

```
print *, ">>", TOSTR(12,"(i4)"), "<<"
produces >>  12<<
```

With integers, TOSTR can also generate leading zeros, which is useful for auto-generating file names or variable names. In such cases, the number of leading zeros is determined by the second optional **integer** parameter. This integer sets the template for the leading zeros, the maximum string. The exact value is unimportant, only the number of digits is used.

For example,

```
print *, ">>", TOSTR(10, 100), "<<"
produces >>010<<
```

```
print *, ">>", TOSTR(10, 999), "<<"
also produces >>010<<
```

```
print *, "File_" // TOSTR(10, 10000) // ".txt"
produces File_00010.txt
```

Examples of arrays

It is possible to convert numeric arrays to their string representation:

```
real, dimension(6) :: RARR = [0.1,0.2,0.3,0.4,0.5,0.6]
.....
print *, ">>", TOSTR(RARR), "<<"
produces > 0.100000001 0.200000003 0.300000012 0.400000006 0.500000000 0.600000024<<
```

Fortran format statement is also accepted for arrays:

```
real, dimension(6) :: RARR = [0.1,0.2,0.3,0.4,0.5,0.6]
.....
print *, ">>", TOSTR(RARR,"(f4.2)"), "<<"
produces >> 0.10 0.20 0.30 0.40 0.50 0.60<<
```

It is possible to use array slices and array constructors with implicit do:

```
print *, ">>", TOSTR(RARR(1:4)), "<<"
print *, ">>", TOSTR( (/(RARR(i), i=1,4)/) ), "<<"
both produce >> 0.100000001 0.200000003 0.300000012 0.400000006<<
```

or using the newer format with square brackets:

```
print *, ">>", TOSTR( [(RARR(i), i=1,4), 200.1, 400.5] ), "<<"
produces >> 0.100000001 0.200000003 0.300000012 0.400000006 200.100006 400.500000<<
```

the same with format:

```
print *, ">>", TOSTR( [(RARR(i), i=1,4), 200.1, 400.5], "(f9.3)" ), "<<"
produces >> 0.100 0.200 0.300 0.400 200.100 400.500<<
```

The subroutine TOSTR is useful because it allows to change such confusing old-style Fortran string constructions as this

```
!print new gene pool. First make file name      !BSA 18/11/13
if (gen < 10) then
  write(gen1,2902) "gen-0000000",gen
else if (gen < 100) then
  write(gen1,2903) "gen-0000000",gen
else if (gen < 1000) then
  write(gen1,2904) "gen-000000",gen
else if (gen < 10000) then
  write(gen1,2905) "gen-00000",gen
else if (gen < 100000) then
  write(gen1,2906) "gen-0000",gen
else if (gen < 1000000) then
  write(gen1,2907) "gen-000",gen
else if (gen < 10000000) then
  write(gen1,2913) "gen-00",gen
else if (gen < 100000000) then
  write(gen1,2914) "gen-0",gen
else
  write(gen1,2915) "gen-",gen
end if

if (age < 10) then
  write(gen2,2920) "age-0000",age
else if (age < 100) then
  write(gen2,2921) "age-000",age
else if (age < 1000) then
  write(gen2,2922) "age-00",age
else if (age < 10000) then
  write(gen2,2923) "age-0",age
else
  write(gen2,2924) "age-",age
end if

write(gen3,2908)gen1,"-",gen2

if (expmt < 10) then
  write(string104,2901)"HED24-",MMDD,runtag,"-E0",expmt,"-o104-genepool-",gen3,".txt"
else
  write(string104,2910)"HED24-",MMDD,runtag,"-E",expmt,"-o104-genepool-",gen3,".txt"
end if
```

to a much shorter and clear like this:

```
!print new gene pool. First make file name      !BSA 18/11/13
string104 = "HED24-" // trim(MMDD) // trim(runtag) // "-E0" // &
            TOSTR(expmt,10) // "-o104-genepool-" // &
            "gen-" // TOSTR(gen, 10000000) //  "-" // &
            "age-" // TOSTR(age, 10000) // f_exten
```

3.2 Subroutines: STDOUT and STDERR

These subroutines output arbitrary text to the terminal, either to the standard output and standard error. While it seems trivial (standard Fortran print *, or write() can be used), it is still good to have a dedicated standard subroutine for all outputs as we can then easily modify the code to use Matlab/R API to work with and run models from within these environments, or use a GUI window (the least necessary feature now, but may be useful if the environment is used for teaching in future). In such cases we will then implement a specific dedicated output function and just globally swap STDOUT with something like R_MESSAGE_PRINT or X_TXTGUI_PRINT.

STDOUT/STDERR accept an arbitrary number of string parameters, which just represent messages placed to the output. Each parameter is printed on a new line. Trivial indeed:)

> **Important**
> It is useful to have two separate subroutines for stdout and stderr as they could be easily separated (e.g. redirected to different files). Redirection could be done under Windows/Linux terminal in such a simple way:
> `model_command.exe 1>output_file_stdout 2>output_file_stderr`
> Here STDOUT is redirected to output_file_stdout, STDERR, to output_file_stderr.

Examples

```
call STDOUT("----------------------------------------------------",&
             ch01 // " = " // ch02 // TOSTR(inumber) // " ***", &
             ch10 // "; TEST NR= " // TOSTR(120.345), &
             "Pi equals to = " //  TOSTR(realPi, "(f4.2)"), &
             "----------------------------------------------------")
```

The above code just prints a message. Note that TOSTR function is used to append numerical values to the text output (unlike standard write where values are separated by commas).

3.3 Function: CLEANUP

CLEANUP Removes all spaces, tabs, and any control characters from the input string. It is useful to make sure there are no trailing spaces in fixed Fortran strings and no spaces in file names.

Example:

```
print *, ">>", CLEANUP("This is along string blablabla"), "<<"
produces >>Thisisalongstringblablabla<<
```

3.4 Subroutine: RANDOM_SEED_INIT_SIMPLE

RANDOM_SEED_INIT_SIMPLE is called without parameters and just initialises the random seed for the Fortran random number generator. But note that the module `BASE_RANDOM` contains a much better subroutine `RANDOM_SEED_INIT` that is also suitable for parallel processing systems (`RANDOM_SEED_INIT_SIMPLE` **cannot** be used in parallel calculations).

Example

```
call RANDOM_SEED_INIT
```

4 Module: CSV_IO

4.1 Overview

This module contains subroutines and functions for outputting numerical data to the CSV (Comma Separated Values) format (RFC4180, CSV format). There are now only routines for data output to CSV, not (yet?) for input as we don't input much data.

The typical workflow for output in CSV file format is like this:

- **CSV_OPEN_WRITE** - physically open CSV file for writing;
- **CSV_HEADER_WRITE** - physically write optional descriptive header (header is just the first line of the CSV file);
- do — start loop (1) over records (rows of data file)
  do — start loop (2) over values within the same record
  CSV_RECORD_APPEND - produce record of data values of different types, append single values, arrays or lists, usually in loop(s)
  end do — end loop (2)
  CSV_RECORD_WRITE - physically write the current record of data
  to the output file.
- end do — end loop (1) — go to producing the next record;
- **CSV_CLOSE** - physically closes the output CSV file.

Thus, subs ending with **_WRITE** and **_CLOSE** do physical write.

This module is most suited at this moment for CSV file *output* rather than input.

This module widely uses **optional arguments**. They may or may not be present in the function/subroutine call. If not all parameters are passed, so called *named parameters* are used. That is, the name of the parameter(s) within the function is explicitly stated when the function/subroutine is called.

For example, `GET_FREE_FUNIT` has its both parameters optional (`max_funit` and `file_status`), it can be called in the standard way as below:

```
intNextunit = GET_FREE_FUNIT(200, logicalFlag)
```

It can lack any parameter:

```
intNextunit = GET_FREE_FUNIT()
```

If the first optional parameter is absent, `GET_FREE_FUNIT` is called as here:

```
intNextunit = GET_FREE_FUNIT(file_status=logicalFlag)
```

If both parameters present but swapped in order, it should be

```
intNextunit = GET_FREE_FUNIT(file_status=logicalFlag, max_funit=200)
```

of course, it can also be used this way:

```
intNextunit = GET_FREE_FUNIT(max_funit=200, file_status=logicalFlag)
```

Important
The standard way of using subroutine parameters (without explicitly setting their names) when calling subroutine works only when their are not missing and their order remains the same as in the subroutine declaration. When a function / subroutine has many parameters and optional are interspersed with mandatory, *it is probably just safer to use named parameters anyway*.

Files can be referred either by unit or by name, but unit has precedence (if both a provided, unit is used). There is also a derived type **csv_file** that can be used as a single file handle. If csv_file object is defined, the file name, unit and the latest operation success status can be accessed as `%name`, `%unit`, `%status` (e.g. `some_file%name`, `some_file%unit`).

The physical file operation error flag, `csv_file_status` is of logical type. It is always an optional parameter.

Here is an example of the data saving workflow:

```
use CSV_IO  ! invoke this module first
........
........
! 1. Generate file name for CSV output
csv_file_append_data_name="data_genomeNR_" // TOSTR(i) // "_" // TOSTR(j) // &
                          "_" // TOSTR(k) // ".csv"
........
! 2. open CSV file for writing
call CSV_OPEN_WRITE (csv_file_append_data_name, csv_file_append_data_unit, &
                     csv_written_ok)
if (.not. csv_written_ok) goto 1000 ! handle possible CSV error
! 3. Write optional descriptive header for the file
call CSV_HEADER_WRITE(csv_file_name = csv_file_append_data_name, &
                      header = header_is_from_this_string, &
                      csv_file_status = csv_written_ok)
........
........
! 4. Generate a whole record of variable (column) names
record_csv="" ! but first, prepare empty record string
call CSV_RECORD_APPEND(record_csv,["VAR_001", ("VAR_" // TOSTR(i,100),i=2,Cdip)])
! 5. physically write this variable header record to the file
call CSV_RECORD_WRITE (record=record_csv, &
                       csv_file_name=csv_file_append_data_name,&
                       csv_file_status=csv_written_ok)
if (.not. csv_written_ok) goto 1000 ! handle possible CSV error
........
........
! 6. Now we can write records containing actual data values, we do this
!    in two do-cycles
CYCLE_OVER_RECORDS: do l=1, Cdip
  ! 7. Prepare an empty string for the current CSV record
  record_csv=""
  CYCLE_WITHIN_RECORD: do m=1, CNRcomp
    ....
    ! do some calculations...
    .....
    .....
    ! 8. append the next value (single number: genomeNR) to the current record
    call CSV_RECORD_APPEND ( record_csv, genomeNR(l,m) )
    .....
  end do CYCLE_WITHIN_RECORD
  ! 9. physically write the current record
  call CSV_RECORD_WRITE ( record=record_csv, &
                          csv_file_name=csv_file_append_data_name,&
                          csv_file_status=csv_written_ok )
  if (.not. csv_written_ok) goto 1000 ! handle possible CSV error
  .......
end do CYCLE_OVER_RECORDS
! 10. close the CSV file when done
call CSV_CLOSE( csv_file_name=csv_file_append_data_name, &
                csv_file_status=csv_written_ok )
if (.not. csv_written_ok) goto 1000 ! handle possible CSV error
```

Although, there is a wrapper for saving the whole chunk of the data at once. A whole array or matrix (2-dimensional table) can be exported to CSV in a single command:

```
! save the whole matrix/array d_matrix to some_file.csv
call CSV_MATRIX_WRITE(d_matrix, "some_file.csv", fstat_csv)
if (.not. fstat_csv) goto 1000
```

4.2 Subroutine: CSV_OPEN_WRITE

Open CSV file for writing. May have two forms:

(1) either get three parameters:

```
character (len=*) :: csv_file_name  ! file name
integer :: csv_file_unit            ! file unit
logical :: csv_file_status          ! optional status flag, TRUE if operation
                                    !   successful
```

(2) get the (single) file handle object of the derived type `csv_file`

```
type(csv_file), intent(inout) :: csv_file_handle  ! file handle object
```

Example

```
type(csv_file) :: file_occ       ! declare file handle object
........
call CSV_OPEN_WRITE(file_occ)    ! use file handle object
........
call CSV_OPEN_WRITE(file_name_data1, file_unit_data1, fstat_csv) ! old style
 if (.not. fstat_csv) goto 1000
```

4.3 Subroutine: CSV_CLOSE

Closes a CSV file for reading or writing. May have two forms:

(1) either get three optional parameters:

```
character (len=*) :: csv_file_name  ! file name
integer :: csv_file_unit            ! file unit
logical :: csv_file_status          ! optional status flag, TRUE if operation
                                    !   successful
```

Important
At least **file name** or **unit** should be present in the subroutine call.

(2) get one file handle object of the derived type `csv_file`

```
type(csv_file), intent(inout) :: csv_file_handle  ! file handle object
```

Example

```
type(csv_file) :: file_occ          ! declare file handle object
........
call CSV_CLOSE(file_occ)            ! use file handle object
........
call CSV_CLOSE(csv_file_name=file_name_data1, &   ! old style
               csv_file_status=fstat_csv)
 if (.not. fstat_csv) goto 1000
```

4.4 Subroutine: CSV_HEADER_WRITE

Writes an optional descriptive header to a CSV file. The header should normally be the first line of the file.

May have two forms:

(1) either get four parameters, only the header is mandatory, but the file must be identified by name or unit:

```
character (len=*) :: csv_file_name  ! file name
integer :: csv_file_unit            ! file unit
character (len=*) :: header         ! header string
logical :: csv_file_status          ! status flag, TRUE if operation successful
```

Important
At least **file name** or **unit** should be present in the subroutine call.

---

(2) get two parameters including the header string and the file handle object of the type `csv_file`

```
character (len=*) :: header         ! mandatory CSV file header
type(csv_file) :: csv_file_handle   ! file handle object
```

Example

```
call CSV_HEADER_WRITE(csv_file_name=FILE_NAME_CSV1, &
      header="Example header. Total " // TOSTR(CSV_RECORD_SIZE(record_csv)) // &
      " columns of data.", csv_file_status=fstat_csv)
if (.not. fstat_csv) goto 1000
```

Here CSV file header is generated from several components, including the `CSV_RECORD_SIZE` function to count the record size.

4.5 Function: GET_FILE_UNIT

Returns file unit associated with an existing open file name, if no file unit is associated with this name (file is not opened), return unit=-1 and error status

Input parameters:

```
character (len=*) :: csv_file_name    ! mandatory file name
logical :: csv_file_status            ! optional status flag, TRUE if operation
                                      !   successful
```

Output parameter (function value):

```
integer :: csv_file_unit              ! unit associated with open file name
```

Example

```
file_unit = GET_FILE_UNIT(file_name)
```

4.6 Function: GET_FREE_FUNIT

Returns the next free/available Fortran file unit number. Can optionally search until a specific maximum unit number.

Input parameters, optional:

```
logical :: file_status              ! operation success status
integer :: max_funit                ! maximum unit to search
```

Output parameter (function value):

```
integer :: file_unit                ! the first free/available file unit
```

Important
When optional input parameters are absent, the function uses a hardwired maximum unit number, possibly depending on the computer platform and compiler used.

Example

```
restart_file_unit_27 = GET_FREE_FUNIT()
```

4.7 Function: CHECK_UNIT_VALID

Checks if file unit is valid, that is within the allowed range and doesn't include standard input/output/stderr units. The unit should not necessarily be linked to any file or be an open file.

Input parameter:

```
integer :: file_unit                ! Fortran file unit to check
```

Output parameter (function value):

```
logical :: file_status              ! gets TRUE if the unit is valid
```

Example

```
if (.not. CHECK_UNIT_VALID(csv_file_unit)) then
      csv_file_unit=GET_FREE_FUNIT(csv_file_status, MAX_UNIT)
.........
```

In this example, we check if the user provided unit is valid, if not, get the first available one.

4.8 Function: CHECK_FILE_OPEN

Checks if a file is currently open, can optionally determine the Fortran unit associated with an open file (returns -1 if it is not open). Input parameters can be either raw form (file name or unit) or csv_file object. Optional `csv_file_status` can determine if the check proceeded without error (=TRUE) there was an error when trying to access the file (=FALSE). Input parameters must be either file name or unit.

Standard (verbose) form:

```
! Calling parameters
character (len=*), optional, intent(in) :: csv_file_name  ! file name to check
integer, optional, intent(in) :: csv_file_unit            !  or unit to check
logical, optional, intent(out) :: csv_file_status         ! error status
integer, optional, intent(out) :: get_csv_file_unit       ! obtain file unit of
                                                          !  an open file
```

File object form:

```
type(csv_file) :: csv_file_handle
```

Output of the function is logical type, returns TRUE if the file is currently opened, FALSE otherwise.

Examples:

```
if (.not. CHECK_FILE_OPEN("file_out.csv")) call OPEN_OUTPUT
...
if (CHECK_FILE_OPEN(csv_file_unit=12)) goto 100
...
file_is_open = CHECK_FILE_OPEN(csv_file_name="data_out.csv", &
                               get_csv_file_unit=fileunit, &
                               csv_file_status=error_flag)
```

```
type (csv_file) :: output_handle
...
if (CHECK_FILE_OPEN(output_handle)) then
...
```

4.9 Subroutine: CSV_RECORD_APPEND

Appends one of the possible data objects to the current CSV record. Data objects could be either a single value (integer, real with single or double precision, character string) or a one-dimensional array of the above types or still an arbitrary length list of the same data types from the above list.

4.9.1 Overview

The first parameter of the subroutine is always character string record:

```
character (len=*) :: record          ! character string record to append data
```

The other parameters may be of any of thee following types: `integer (kind=4)`, `real(kind=4)`, `real(kind=8)`, `character` string.

Important
The record keeping variable can be either fixed length string or an allocatable string. But it should fit the whole record. This might be a little bit tricky if record is allocatable as `record_string=""` allocates it to an empty string. A good tip is to use the `repeat` function in Fortran to allocate the record string to the necessary value, e.g. `record= repeat(" ", MAX_RECORD)` will produce a string consisting of `MAX_RECORD` blank characters. `record` should not necessarily be an empty string initially, it could be just a whole blank string.

4.9.2 Examples

Append a single string to the current record:

```
call CSV_RECORD_APPEND(record_csv, "ROW_NAMES")
```

Append a single value (any of the supported types) to the current record:

```
call CSV_RECORD_APPEND(record_csv, value)      ! some variable of supported type
call CSV_RECORD_APPEND(record_csv, 123.5_8)    ! double precision literal value
```

Append a list of values (any one of the supported types) to the current record:

```
call CSV_RECORD_APPEND(record_csv, fish, age, stat4, fecund)
```

Append an array slice (any of the supported types) to the current record:

```
call CSV_RECORD_APPEND(record_csv, RARR(1:4))
```

Append an array using old-style array constructor with implied do (any of the supported types) to the current record:

```
call CSV_RECORD_APPEND(record_csv,(/(RARR(i), i=1,6)/))
```

Append an array using new-style array constructor (square brackets) with implied do plus two other values (all values can have any of the supported types but should have the same type) to the current record:

```
call CSV_RECORD_APPEND(record_csv, [(RARR(i), i=1,4), measur1, age(fish)])
```

Append integers from 1 to 10 to the current record (using implied do):

```
call CSV_RECORD_APPEND(record_csv, [(i,i=1,10)])
```

Append a string, an array of strings with implied do and finally another string to the record. This example shows how variable (column) names could be generated:

```
call CSV_RECORD_APPEND(record_csv,["ROW_NAME",("VAR_" // TOSTR(i,1000),i=1,1000),"STATUS"])
```

Important
On some compilers (e.g. Oracle Solaris Studio f95 v.12 but not GNU gfortran version >5), all strings within the array constructor must explicitly have the same length, otherwise the compiler issues an error. In gfortran (v>5, the first occurrence of the string (e.g. the first iteration of the implied do loop) defines the default length and all extra characters are just silently dropped. The behaviour of other compilers and their versions may differ.

4.10 Function: CSV_RECORD_SIZE

Counts the number of values in a CSV record.

Input parameters:

```
character (len=*) :: record           ! mandatory CSV record
```

Function value: an integer

```
integer :: csv_record_size
```

Example

```
print *, "This record is: ", CSV_RECORD_SIZE(record_csv), " columns."
```

4.11 Function: CSV_FILE_LINES_COUNT

Counts the number of lines in an existing CSV file. If file cannot be opened or file error occurred, then issues the value -1

Input parameters:

```
character (len=*) :: csv_file_name  ! The name of the existing file
logical :: csv_file_status          ! optional file operation status, TRUE if
                                    !   file operations were successful.
```

Function value: an integer

```
integer :: csv_file_lines_count     ! number of lines in file, -1 if file error
```

Can actually calculate the number of lines in any text file. Does not distinguish header or variable names lines in the CSV file and does not recognize CSV format.

Example

```
print *, "File ", CSV_FILE_LINES_COUNT("test_file.csv", succ_flag), "lines."
```

4.12 Subroutine: CSV_RECORD_WRITE

Physically writes a complete record of data to a CSV file. A record is a single row of data in the file.

This subroutine has two forms:

(1) it can either accept three parameters:

```
character (len=*) :: csv_file_name          ! file name
integer :: csv_file_unit                    ! file unit
character (len=*)  :: record                ! current CSV record (mandatory)
logical :: csv_file_status                  ! optional operation status, TRUE if
                                            !  success
```

> **Important**
> The file to write the current record can be referred either by name or unit. So one of them must be present in the subroutine call.

(2) get the CSV record and the (single) file handle object of the derived type `csv_file`

```
character (len=*)  :: record                ! current CSV record (mandatory)
type(csv_file) :: csv_file_handle           ! file handle object
```

Example

```
call CSV_RECORD_WRITE(csv_record, file_cop)                 ! write current record
call LOG_MSG("Physically wrote record " // TOSTR(a) // &    ! report this in some
             " to the file " // file_cop%name // &          ! logging subroutine.
             ", write status =" // TOSTR(file_cop%status))
```

Note, that file handle object is used in the above example.

4.13 Subroutine: CSV_MATRIX_WRITE

Writes a matrix of real (kind 4 or 8), integer or string values to a CSV data file. This is a shortcut allowing to write data in a single code instruction. This subroutine works either with a two-dimensional matrix or one-dimensional array (vector). The behaviour is a little different in these cases.

4.13.1 Two-dimensional matrix

It gets the following parameters: (1) two-dimensional data matrix (of any supported type), (2) mandatory name of the output file; (3) optional vector of column names. If the column name vector is shorter than the "column" dimension of the data matrix, the remaining columns get "COL_XXX" names, where XXX is the consecutive column number (so they are unique). and (4) optional logical file operation success status.

```
[any supported], dimension(:,:) :: matrix ! data object, array or 2-d matrix
character (len=*) :: csv_file_name        ! file name for output
character, dimension(:) :: colnames       ! optional array of column names
logical :: csv_file_status                ! operation status, TRUE if success
```

Example

```
real, dimension(1:100,1:30 ) :: MATRIX
character (len=8), dimension(1:10) :: NAMES = ["MEAS_001","MEAS_002","MEAS_003",&
    "MEAS_004","MEAS_005","MEAS_006","MEAS_007","MEAS_008","MEAS_009","MEAS_010"]
....
! save data with column names, the first ten names are taken from the NAMES
!    string array, the remaining ones are autogenerated
call CSV_MATRIX_WRITE(matrix=MATRIX, colnames=NAMES,
                      csv_file_name="data_file.csv", csv_file_status=fstat_csv)
if (.not. fstat_csv) goto 1000

! save data without column names
call CSV_MATRIX_WRITE(matrix=MATRIX, csv_file_name="data_file.csv",
                      csv_file_status=fstat_csv)
if (.not. fstat_csv) goto 1000
```

Higher-rank arrays (with more than two dimensions)[2] can be saved into CSV files using array slices, for example:

```
real, dimension(100,300,99) :: M3   ! Declare a 3D matrix M3
....
do i=lbound(M3,3), ubound(M3,3)     ! Cycle over the third index, min - max

  ! Save separate slices of M3 into individual files file_01.csv .. file_99.csv
  call CSV_MATRIX_WRITE(matrix=M3(:,:,i), &
                        colnames=NAMES,   &
                        csv_file_name="file_" // TOSTR(i,10) // ".csv", &
                        csv_file_status=flag)

end do
```

4.13.2 One-dimensional arrays

With one-dimensional array (vector), the subroutine gets (1) the array, (2) output file name, (3) logical parameter pointing if the array is saved "vertically" (as a single column, if `TRUE`) or "horizontally" (as a single row, if `FALSE`). If the `vertical` parameter is absent, the default `TRUE` (i.e. "vertical" data output) is used. There is also an alias to this subroutine, **`CSV_ARRAY_WRITE`**.

```
[any supported], dimension(:) :: array    ! data object, array
character (len=*) :: csv_file_name        ! file name for output
logical :: vertical                       ! optional parameter defining how one-
                                          !   dimensional array is saved
logical :: csv_file_status                ! operation status, TRUE if success
```

Example

```
! Here the data will be written into a single row of values
call CSV_MATRIX_WRITE (ARRAY, "data_file.csv", .FALSE., fstat_csv)
 if (.not. fstat_csv) goto 1000
```

Tip

In the simplest cases, with only the data object and the file name, `CSV_MATRIX_WRITE` can be used with a two-dimensional matrix or one-dimensional array in the same way (it's convenient during debugging):

[2] CSV_IO code could be modified to save higher-rank arrays if this function is needed

```
real, dimension(1:100,1:20) :: MatrixX     ! Matrix, two dimensional
real, dimension(1:100) :: Array_Y          ! Array, one-dimensional
.......
.......
call CSV_MATRIX_WRITE(MatrixX, "file_matrixx.csv")       ! write 2-d matrix
call CSV_MATRIX_WRITE(Array_Y, "file_array_y.csv")       ! write 1-d array
```

4.14 Derived type: csv_file

This type is used as a unitary file handle object. It has the following structure:

```
type, public :: csv_file
  character (len=MAX_FILENAME) :: name  ! The name of the file
  integer :: unit = -1                  ! Fortran unit associated with the file
  logical :: status = .TRUE.            ! success flag for the latest operation
end type csv_file
```

If `csv_file` object is defined, the file name, unit and the latest operation success flag can be accessed as `%name`, `%unit`, `%status` (e.g. `some_file%name`, `some_file%unit`).

4.14.1 Basic Example

```
type(csv_file) :: file_occ              ! define the file handle object
....
file_occ%name="some_name.txt"           ! set file name value
....
call CSV_OPEN_WRITE(file_occ)           ! Open file for writing
....
call CSV_CLOSE(file_occ)                ! Close file
```

4.14.2 Arrays of structures

This derived type can be also used as an array. An example below shows how can this be done.

```
type(csv_file), dimension(:), allocatable :: file_ABM ! Define allocatable array
........                                              !   of file handle objects
allocate(file_ABM(modulators))                        ! Allocate this array
........
! now, use the array to handle many files of the same type
do j=1, modulators
  file_ABM(j)%name = "file_no_" // TOSTR(j,10) // ".csv"  ! Set file handle (j)
  call CSV_OPEN_WRITE(file_ABM(j))                        !   and use it
end do
```

Important

The file name is set as a standard **non-allocatable** fixed string because allocatable strings may not be supported on all compiler types and versions. Notably, older GNU gfortran (prior to v.5) does not allow allocatable strings in derived types. Currently, `MAX_FILENAME=255` (can be changed in the code). There is one consequence of using fixed strings: you may have to use the Fortran `trim()` function to cut off trailing blanks if strings are concatenated. E.g. do `file_name=trim(String1) //trim(String2)` instead of `file_name=String1 //String2` or use `file_name=CLEANUP(String1 //String2)` to remove all blank and control characters.

5 Module BASE_RANDOM

This module contains subroutines for generating random numbers. However, the code of this module depends on the platform and compiler used. The build system generates the appropriate header file automatically.

5.1 RANDOM_SEED_INIT

Initialise the random seed for random number generation. This module uses an improved random seed generation algorithm that uses the system entropy pool on Unix systems and XOR of the current time and PID on Windows. Therefore, it is **safe** for use on **parallel processing systems**. Normally has no parameters.

```
call RANDOM_SEED_INIT()
```

`RANDOM_SEED_INIT` can optionally return the current (calculated) seed as two parameters: integer dimension of the seed array `n_here` and the array itself `seed_here`. This, however, is useful only for debugging.

```
integer :: dbg_seed_size                        ! depends on compiler/platform
integer, dimension (12) :: dbg_seed_array       ! ... can be 12 or 2 on x86
...
call RANDOM_SEED_INIT(dbg_seed_size, dbg_seed_array)
print *, "Seed: ", dbg_seed_size, ", array: ", dbg_seed_array(:dbg_seed_size)
```

The seed array size can be different: on GNU gfortran x86 it is **12**, on Intel and Oracle Fortran (both x86) it is **2**.

5.2 RAND_I

Generates a random integer within the range A to B (the two parameters of the function).

```
ipos = RAND_I(1, len(ga_target))
```

5.3 RAND_R4 and RAND_R8

Generates a random real (type 4 or 8). Has no parameters.

```
if ( RAND_R4() < ga_mutationrate ) then
  call mutate(fish(i))
end if
```

5.4 Build details

When **not using** the automatic build system based on GNU make, the module subroutine `RANDOM_SEED_INIT` should be tweaked according to the compiler and platform as follows:

GNU fortran:

```
!*****************************************************************************
! *** NON-PORTABLE CODE BEGIN ***

use ISO_FORTRAN_ENV, only: int64    ! GNU and Intel

implicit none

integer, allocatable :: seed(:)
integer :: i, n, un, istat, dt(8), pid
integer(int64) :: t

! *** NON-PORTABLE CODE END ***
!*****************************************************************************
```

Intel Fortran

```
!***********************************************************************
! *** NON-PORTABLE CODE BEGIN ***

use ISO_FORTRAN_ENV, only: int64  ! GNU and Intel

use IFPORT, only : getpid         ! getpid is an extension defined in IFPORT

implicit none

integer, allocatable :: seed(:)
integer :: i, n, un, istat, dt(8), pid
integer(int64) :: t

! *** NON-PORTABLE CODE END ***
!***********************************************************************
```

Oracle Fortran

```
!***********************************************************************
! *** NON-PORTABLE CODE BEGIN ***

! External Modules not used on Oracle f95, but an include header must be placed

implicit none

integer, allocatable :: seed(:)
integer :: i, n, un, istat, dt(8), pid
integer, parameter :: int64 = selected_int_kind(18) ! define int64
integer(int64) :: t

include "system.inc"  ! Include non-intrinsic library headers for the Oracle f95

! *** NON-PORTABLE CODE END ***
!***********************************************************************
```

The build system based on GNU make does this automatically.

6 Module: LOGGER

This module controls to log different messages during the execution of the program. The format of the messages is configutrable during the run time.

To be done

7 Error trapping modules: ERRORS, ASSERT, EXCEPTIONS

These modules can be used for error trapping and handling.

To be done

8 Module: BASE_STRINGS

This module containing some useful string manipulation functions is borrowed from http://www.gbenthien.net/strings/index.html. The description below is just repeating the official doc file included with the module. Note that there are a couple of utils

(`READLINE`, `WRITEQ`) in this module that work with files. These use the standard Fortran unit to refer for the file and unlike the other modules here are not adjusted (yet) to use the file handle object (`csv_file`).

Fortran Character String Utilities. A collection of string manipulation routines is contained in the module 'strings' found in the file stringmod.f90. To obtain this module as well as some other string utilities, go to the website http://www.gbenthien.net/-strings/index.html. To use the routines in the module 'strings' the user needs to add the statement `use strings` to the top of the program. These routines were developed primarily to aid in the reading and manipulation of input data from an ASCII text file. The routines are described below.

8.1 Subroutine: PARSE

```
SUBROUTINE PARSE(str, delims, args, nargs)
```

This routine was originally designed to separate the arguments in a command line where the arguments are separated by certain delimiters (commas, spaces, etc.). However, this routine can be used to separate other types of strings into their component parts. The first input is a string `str` (e.g., a command line). The second argument is a string `delims` containing the allowed delimiters. For example, `delims` might be the string `" ,"` consisting of a comma and a space. The third argument is a character array `args` that contains on output the substrings (arguments) separated by the delimiters. Initial spaces in the substrings (arguments) are deleted. The final argument is an integer `nargs` that gives the number of separated parts (arguments). To treat a delimiter in `str` as an ordinary character precede it by a backslash (\). If a backslash character is desired in `str`, precede it by another backslash (\\). In addition, spaces that immediately precede or follow another delimiter are not considered delimiters. Multiple spaces or tabs are considered as a single space, i.e., `"a b"` is treated the same as `"a b"`. Backslashes can be removed from an argument by calling the routine `REMOVEBKSL`, i.e.,

```
call REMOVEBKSL(<string>)
```

This routine converts double backslashes (\\) to single backslashes (\).

Example: If the delimiters are a comma and a space (`delims =" ,"`), then the subroutine `PARSE` applied to the string `"cmd arg1 arg\2 arg3"` produces the output:

```
args(1) = cmd
args(2) = arg1
args(3) = arg 2
args(4) = arg3
nargs = 4
```

8.2 Subroutine: COMPACT

```
SUBROUTINE COMPACT(str)
```

This routine converts multiple spaces and tabs to single spaces and deletes control characters.

8.3 Subroutine: REMOVESP

```
SUBROUTINE REMOVESP(str)
```

This routine removes spaces, tabs, and control characters in string `str`.

8.4 Subroutine: VALUE

```
SUBROUTINE VALUE(str, number, ios)
```

This subroutine converts a number string to a number. The argument `str` is a string representing a number. The argument number is the resulting real number or integer (single or double precision). The argument `ios` is an error flag. If `ios` is nonzero, then there was an error in the conversion.

8.5 Subroutine: SHIFTSTR

```
SUBROUTINE SHIFTSTR(str, n)
```

This routine shifts characters in the string `str` by `n` positions (positive values denote a right shift and negative values denote a left shift). Characters that are shifted off the end are lost. Positions opened up by the shift are replaced by spaces.

8.6 Subroutine: INSERTSTR

```
SUBROUTINE INSERTSTR(str, strins, loc)
```

This routine inserts the string strins into the string `str` at position `loc`. Characters in `str` starting at position `loc` are shifted right to make room for the inserted string.

8.7 Subroutine: DELSUBSTR

```
SUBROUTINE DELSUBSTR(str, substr)
```

This subroutine deletes the first occurrence of substring `substr` from string `str` and shifts characters left to fill hole.

8.8 Subroutine: DELALL

```
SUBROUTINE DELALL(str, substr)
```

This routine deletes all occurrences of substring `substr` from string `str` and shifts characters left to fill holes.

8.9 Function: UPPERCASE

```
FUNCTION UPPERCASE(str)
```

This function returns a string that is like the string `str` with all characters that are not between a pair of quotes (`" "` or `' '`) converted to uppercase.

8.10 Function: LOWERCASE

```
FUNCTION LOWERCASE(str)
```

This function returns a string that is like the string `str` with all characters that are not between a pair of quotes (`" "` or `' '`) converted to lowercase.

8.11 Subroutine: READLINE

```
SUBROUTINE READLINE(nunitr, line, ios)
```

This routine reads a line from unit `nunitr`, ignoring blank lines and deleting comments beginning with an exclamation point(!). The line is placed in the string `line`. The argument `ios` is an error flag. If `ios` is not equal to zero, then there has been an error in the read operation. A negative value for `ios` denotes an end of file.

8.12 Subroutine: MATCH

```
SUBROUTINE MATCH(str, ipos, imatch)
```

This routine finds the delimiter in string `str` that matches the delimiter in position `ipos` of `str`. The argument `imatch` contains the position of the matching delimiter. Allowable delimiters are (), [], { }, <>.

8.13 Subroutine: WRITENUM

`SUBROUTINE WRITENUM(number, string, fmt)`

This routine writes a number to a string. The argument number is a real number or an integer (single or double precision). The number number is written to the character string string with format `fmt` (e.g., `"e15.6"` or `"i5"`).

8.14 Subroutine: TRIMZERO

`SUBROUTINE TRIMZERO(str)`

This subroutine deletes nonsignificant trailing zeroes in a number string `str`. A single zero following a decimal point is allowed. For example, `"1.50000"` is converted to `"1.5"` and `"5."` is converted to `"5.0"`.

8.15 Subroutine: WRITEQ

`SUBROUTINE WRITEQ(unit, name, value, fmt)`

This routine writes a string of the form `"name=value"` to the unit `unit`. Here `name` is the input string name and value is the input number value converted to a string with the format `fmt`. The number value can be a real number or an integer (single or double precision).

8.16 Function: IS_LETTER

`FUNCTION IS_LETTER(ch)`

This function returns the logical value `.TRUE.` if the input character `ch` is a letter (a–z or A–Z). It returns the value `.FALSE.` otherwise.

8.17 Subroutine: IS_DIGIT

`FUNCTION IS_DIGIT(ch)`

This function returns the logical value `.TRUE.` if the input character `ch` is a digit (0–9). It returns the value `.FALSE.` otherwise.

8.18 Subroutine: SPLIT

`SUBROUTINE SPLIT(str, delims, before, sep)`

This routine uses the first occurrence of a character from the string `delims` in the string `str` to split the string into two parts. The portion of `str` before the found delimiter is output in before; the portion of `str` after the found delimiter is output in `str` (`str` is left justified). The output character `sep` (optional) contains the found delimiter. To treat a delimiter in `str` as an ordinary character precede it by a backslash (\). If a backslash is desired in `str`, precede it by another backslash (\\). Repeated applications of `SPLIT` can be used to parse a string into its component parts. Backslashes can be removed by calling the routine REMOVEBKSL, i.e., `call REMOVEBKSL(string)`

9 IEEE Arithmetics

9.1 Overview

The model can now use the IEEE arithmetic modules. They allow exact control of the CPU math features and exceptions caused by invalid calculations, such as dividion by zero, overflow, underflow etc. A potential issue is that they have an optional status in the Fortran standard, so compilers do not have to implement them, although many do.

> **Important**
> IEEE arithmetic and exceptions are fully described in chapter 14 of this book: Adams, et al., 2009 *The Fortran 2003 Handbook*. Springer.

For example, Intel Fortran implements intrinsic IEEE arithmetics modules. GNU Fortran does not implement them untile version 5.[3] However, there are external (non-intrinsic) IEEE modules for gfortran on the x86 (support both 32 and 64 bit) that are included into the **HEDTOOLS** bundle.

> **Important**
> the **fimm** HPC cluster, where calculations are normally performed, has GNU Fortran 4.8.1 and will require non-intrinsic IEEE modules. It also has the Intel Fortran which has built-in (intrinsic) IEEE modules though.

9.2 IEEE Exceptions

There are several exception conditions:

- `IEEE_DIVIDE_BY_ZERO`
- `IEEE_INEXACT`
- `IEEE_INVALID`
- `IEEE_OVERFLOW`
- `IEEE_UNDERFLOW`
- `IEEE_USUAL` (An array of three exceptions `IEEE_OVERFLOW`, `IEEE_DIVIDE_BY_ZERO`, `IEEE_INVALID`)
- `IEEE_ALL` (An array of five exceptions `IEEE_OVERFLOW`,`IEEE_DIVIDE_BY_ZERO`, `IEEE_INVALID`, `IEEE_UNDERFLOW`, `IEEE_INEXACT`)

Normally, if the program encounters invalid arithmetic calculations, then it should crash or at least report the problem. Otherwise, correctness of calculations is not guaranteed. By default, many compilers just **ignore** invalid calculations (even many cases of division by zero, NaN[4] generation etc.).

In most cases NaNs and other invalid arithmetics strongly point to a bug. It is therefore wise to turn halting ON by default in model calculations (unlike normal utility software that should ideally never crash on trivial math errors).

Turning arithmetic exception halting ON during the compile time requires specific compiler options.

Compiler	**option**	**example**
GNU GCC	`-ffpe-trap`	`-ffpe-trap=zero,invalid,overflow,underflow`
Intel Fortran	`-fpe (/fpe)`	`-fpe0` (`/fpe:0` on Windows)
Solaris Studio	`--ftrap`	`--ftrap=invalid,overflow,division`

The IEEE module `IEEE_EXCEPTIONS` allows to control halting during the run time. For example, it is cool to switch halting ON in specific troublesome parts of the code that can normally result in invalid calculations (division by zero, invalid, inexact etc.) and control each such occurrence specifically (e.g. provide a subroutine handling and fixing the calculations).

Halting the program that encounters specific condition is controlled via `IEEE_GET_HALTING_MODE` subroutine (returns logical parameter `IEEE_DEF_MODE`). For example, for `IEEE_INVALID` it is:

[3] It was because GNU compiler collection is made for portability and supports many different processor architectures in addition to the most common x86 and implementation of IEEE modules is highly dependent on the CPU type and features.
[4] "Not a Number," a wrong arithmetic value that is not equal to itself, can result from many math errors

```
call IEEE_GET_HALTING_MODE(IEEE_INVALID, IEEE_DEF_MODE)
```

It is also possible to set specific halting mode for specific condition. For example, to set halting ON (execution termination) on invalid arithmetic do this:

```
call IEEE_SET_HALTING_MODE(IEEE_INVALID, .TRUE.)  ! Will halt on IEEE_INVALID
```

Here is an example:

```
...
! Invoke IEEE Arithmetics:
! use, non_intrinsic :: IEEE_EXCEPTIONS ! if gfortran v<5

! We normally use included auto-generated wrapper for the module
include "IEEE_wrap.inc"

IMPLICIT NONE

REAL    r,c,C0,Ap,Vc,Ke,Eb
REAL    FR1,FR2,F1,FDER

....

logical :: IEEE_MATH_FLAG, IEEE_DEF_MODE  ! values for IEEE math modules

call IEEE_GET_HALTING_MODE(IEEE_INVALID, IEEE_DEF_MODE) ! Get default halting
call IEEE_SET_HALTING_MODE(IEEE_INVALID, .FALSE.)       ! NO halting from here!

...

FR2=LOG(ABS(C0)*Ap*Vc)
FR1=LOG(((Ke+Eb)/Eb)*r*r*EXP(c*r))
F1 = FR1-FR2
FDER = c + 2./r

call IEEE_GET_FLAG(IEEE_INVALID, IEEE_MATH_FLAG) ! Get the error flag
if(IEEE_MATH_FLAG) then
  ! if IEEE exception is signalled, we cannot relay on the calculations
  ! Report the error: remember there is no halting now, the program won't stop
  write(10,*) "IEEE exception in DERIV ", r,F1,FDER,c,C0,Ap,Vc,Ke,Eb
  ! We also have to fix the calculations, e.g. equate some values to zero
  r=0.; F1=0.; FDER=0.
  call IEEE_SET_FLAG(IEEE_INVALID, .FALSE.) ! Set the error flag back to FALSE
end if

...

call IEEE_SET_HALTING_MODE(IEEE_INVALID, IEEE_DEF_MODE) ! Set default halting

END SUBROUTINE DERIV
```

9.3 Implementation details

We use an automatic build system (see below) which normally keeps track of the compiler and its version and IEEE modules support, there is no need to include `use, intrinsic (or non_intrinsic) ::IEEE_EXCEPTIONS` and tweak it manually depending on the compiler support. The build system automatically generates the correct include file `IEEE_wrap.inc` which should be inserted into the code in place of `use ...` statement:

```
SUBROUTINE DERIV(r,F1,FDER,c,C0,Ap,Vc,Ke,Eb)
```

```
!Derivation of equation for visual range of a predator

! Invoke IEEE Arithmetics:
! use, non_intrinsic :: IEEE_EXCEPTIONS ! if gfortran v<5

! We normally use included auto-generated wrapper for the module
include "IEEE_wrap.inc"

REAL    r,c,C0,Ap,Vc,Ke,Eb
....
```

Without the GNU make-based build system, the rule is simple. Use **non-intrinsic** modules with GNU gfortran version <5.0 [5] and build the modules beforehand:

```
!****************************************************************************
! *** NON-PORTABLE CODE BEGIN ***
use, non_intrinsic :: IEEE_FEATURES
use, non_intrinsic :: IEEE_ARITHMETIC
use, non_intrinsic :: IEEE_EXCEPTIONS
! *** NON-PORTABLE CODE END ***
!****************************************************************************
```

and **intrinsic** modules on GNU gfortran v>5, Intel Fortran or Oracle Fortran:

```
!****************************************************************************
! *** NON-PORTABLE CODE BEGIN ***
use, intrinsic :: IEEE_FEATURES
use, intrinsic :: IEEE_ARITHMETIC
use, intrinsic :: IEEE_EXCEPTIONS
! *** NON-PORTABLE CODE END ***
!****************************************************************************
```

10 Version control: Subversion (SVN)

AHA Repository: https://svn.uib.no/aha-fortran

10.1 Overview

Use version control not only for just managing versions, but also for organising your coding. For example, it would be good to commit changes to the server in pieces involving specific functions or parts of the model that are ready. Use the log messages to describe briefly what has been done.

For example, imagine you have implemented a new sigmpoid function. Then, when it is ready, commit your change to the server with a log message like "New sigmoid function". And only after this go to the next piece of code. Then the versions yoi have will not be haphazard but organised into meaningful pieces. If you did several pieces in different files, e.g. sigmoid function in Hed18.f90 and a new Makefile for building the code, do two commits:

```
svn commit Hed18.f90 -m "New sigmoid function"
...
svn commit Makefile -m "Tweaked makefile, added PGI compiler build"
```

[5] e.g. gfortran on the fimm cluster

10.2 First time setup of the working copy

First time setup of the working copy of the model (working directory):

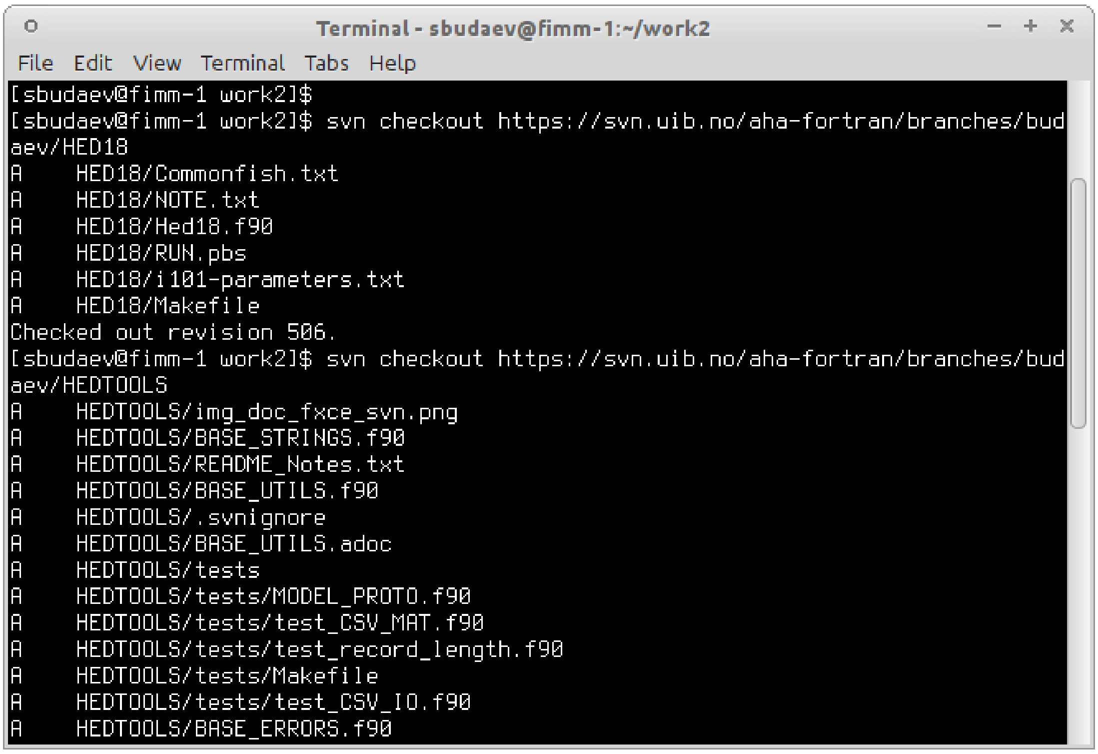
- For a new project (run/experiment etc.), get into the working directory where the model code will reside (`cd`) (possibly make a new directory `mkdir`), and **checkout:** get the model code (one branch, no need to get everything!) from the server with `svn checkout https://path_to_branch` For example,

```
svn checkout https://svn.uib.no/aha-fortran/branches/budaev/HED18
```

This will get the `HED18` into the directory `HED18` within the current working directory. If we use `HEDTOOLS`, it should also be placed here:

```
svn checkout https://svn.uib.no/aha-fortran/branches/budaev/HED18
...
svn checkout https://svn.uib.no/aha-fortran/branches/budaev/HEDTOOLS
```

So, we now get `HED18` and `HEDTOOLS` in our working directory.



10.3 Standard workflow

Now you can work within this directory. This is the standard workflow.

- **update** code from the server: `svn up`
- edit the code using any favoured tools, build, run model etc. . .
- **commit,** when ready (e.g. when a new has been implememnted): `svn commit`

`commit` will ask you to provide a short descriptive log message. It will run the standard text editor for this by default (can be configured). But you can provide such a message just on the command line with the `-m` option:

```
svn commit Hed18.f90 -m "New sigmoid function"
```

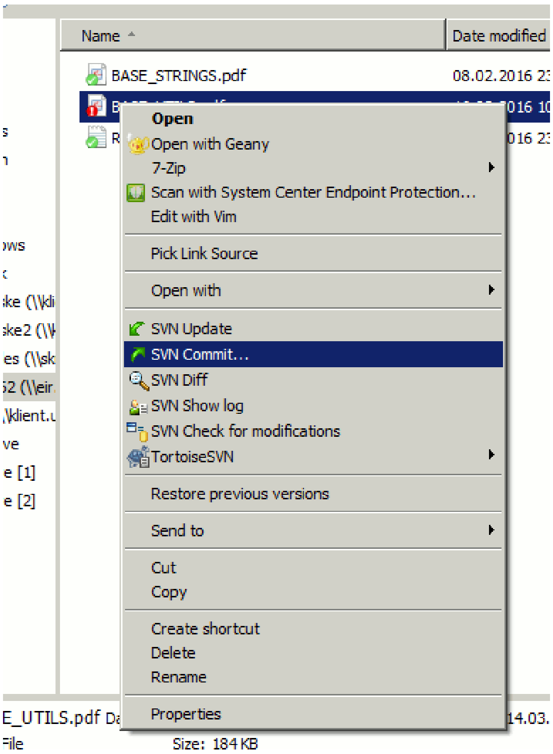
Both `update` and `commit` can be done for the working directory as well as for specific file. E.g. to commit only the model code `Hed18.f90` do:

```
svn commit Hed18.f90
```

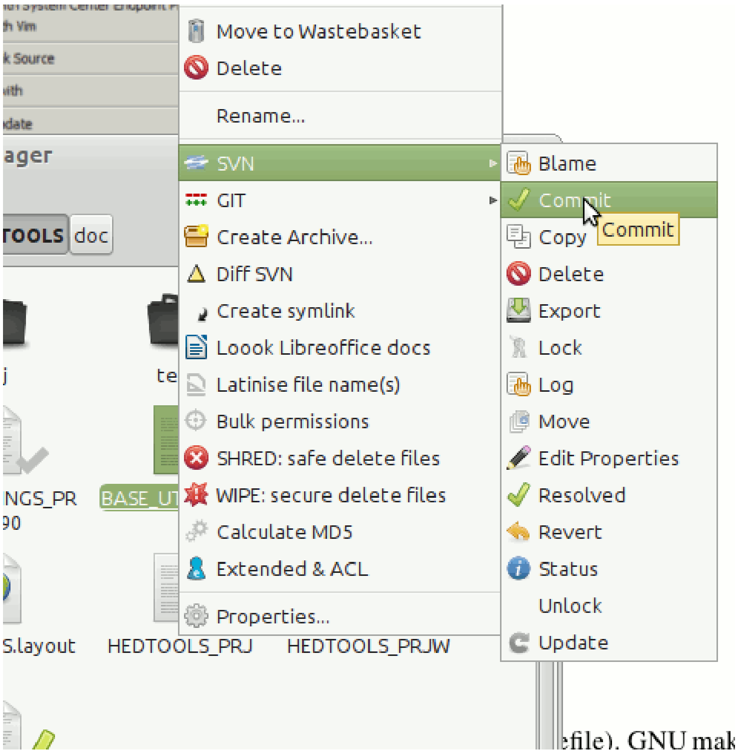
Both `update` and `commit` can be performed within any subdirectory of the working copy. In such cases they are limited to this subdirectory only.

10.4 GUI Tools

Using the GUI tools like TortoiseSVN is similar. With GUI you should just select the appropriate item from the menu list.



Similar GUI tools exist for Linux. For example, there is `thunar-vcs-plugin` (Git and subversion integration into the Thunar file manager).



11 Build system: GNU make

11.1 Overview

The model currently uses a build system based on GNU make (Makefile). GNU make is an automated system for building source core (in fact, any digital project that requires keeping track of dependencies between multiple components.)

> The make program is intended to automate the mundane aspects of transforming source code into an executable. The advantages of make over scripts is that you can specify the relationships between the elements of your program to make, and it knows through these relationships and timestamps exactly what steps need to be redone to produce the desired program each time. Using this information, make can also optimize the build process avoiding unnecessary steps.
>
> — Mecklenburg R. *Managing Projects with GNU Make*

All the build rules for building the model executable are collected in the Makefile. If the model requires external components (e.g. non-intrinsic IEEE math modules), they will be automatically inserted.

GNU make is good because it works on diverse combinations of platforms and OSs (e.g. Linux and Windows). Some proprietary Unix platforms could supply the vendor's `make` utility that may not be compatible with the GNU `make` (e.g. Oracle Solaris includes its own `make` clone). There might be an option turning on GNU compatibility. But it is better to use the GNU `make` (`gmake` on Solaris) anyway.

11.2 Using make

Most basic things with the standard `Makefile` are simple.

11.2.1 Building and running the model

- Get a short help on the options: `make help`

- Build the model executable using default compiler: `make`
- Force rebuild the model executable with Intel compiler: `make intel`
- Force rebuild the model executable with GNU compiler: `make gnu`
- Run the current model: `make run` (on the fimm HPC cluster, this will automatically start a new batch job)

11.2.2 Cleanup

There are also a few options for deteting the files and data generated by the build process.

- Remove all the data generated by the model `make cleandata`
- Remove all the data files generated by the model run as well as the model executable: `make clean`
- Remove everything generated by the build system and all the data, retain the default state: `make distclean`

11.2.3 Debugging

The environment variable `DEBUG` controls whether the build system produces the debug symbols (-g) or, if NOT defined, speed-optimised machine code (-O3, automatic loop parallelization etc.). To build with debug support just define DEBUG in the manner standard for the platform/OS. For example, on Linux use:

```
$ DEBUG=1 make
```

or (`DEBUG` is now persistent)

```
$ export DEBUG=1
$ make
```

on Windows:

```
O:\WORK\MODEL\HED18>set DEBUG=1
O:\WORK\MODEL\HED18>make
```

or use `DEBUG` as a parameter to make, this works on all platforms:

```
$ make intel DEBUG=1
```

The make system keeps track of all the code components. For example, if only one has been changed, it will recompile only this. It also keeps track of whether IEEE math modules are really necessary and if the intrinsic or non-intrinsic modules are used.

For example, you may have built the model executable (make) and then edited the code of a module a little. Then just issue command to run batch (make run) on fimm. The make system will then automatically determine that the model executable is now out of date and recompile the changed module and build an updated executable, and only after this will start the batch job.

Another example: you just checked-out or updated (e.g. `svn up`) the model source that is tested and known to be bug-free on the fimm cluster. Now you should compile components of the program, (e.g. tweak IEEE math modules), build the executable, and finally start the executable in the cluster's batch job system. All this is done using a single command: `make run`.

```
$ svn update
$ ... some output...
$ make run
```

The system should work the same way on Windows and Linux, including the fimm HPC cluster. By editing the `Makefile` provided, one can easily tweak the behaviour of the build process, e.g. add other modules, change names, compilation options and details etc.

Microsoft Studio, Oracle Solaris Studio and other similar IDEs actually provide their own make systems (e.g. `nmake`, `make` or `dmake`) that work behind the scenes even if the IDE GUI is used.

11.3 Tweaking Makefile

11.3.1 Basic parameters

Here go basic parameters of the Makefile

11.3.2 Build options

Here build variables will be placed.

Important
A good manual on the GNU Make is this book: Mecklenburg, R, 2005, *Managing Projects with GNU Make,* Third edition. O'Reilly. There is also the official GNU Make Manual.

12 Coding style: General guidelines and tips

Work at high level, use these tools, use objects, isolate as much as possible into subroutines

In this way of coding, it becomes more clear what each part of the program is really doing and it is also easier to modify components of the program so that they don't affect other irrelevant components.

```
GENERATIONS_LOOP: do while &
                  (realgen <= GENERATIONS .and. &
                   parents(1)%fitness > 0)

  call sort_by_fitness()

  call selection()

  call mate_reproduce()

  call offspring_fitness()

  call generations_swap()

  realgen = realgen +1

end do GENERATIONS_LOOP
```

Use meaningful labels

Using labels to mark `do..end do`, `if ..end if`, `forall` and other similar constructs may greatly improve the readability of the code and make it more easy to understand, especially if there are many nested loops `if..then..end if` constructs. Here is a couple of examples:

```
GENERATIONS_LOOP: do while &
                  (realgen <= GENERATIONS .and. &
                   parents(1)%fitness > 0)
.....
  realgen = realgen +1

end do GENERATIONS_LOOP
```

```
SELECT_DEVIANT_CLASS: if (dev == 2) then
.....
else if (dev == 3) then SELECT_DEVIANT_CLASS
.....
else if (dev == 4) then SELECT_DEVIANT_CLASS
......
end if SELECT_DEVIANT_CLASS
```

Use whole-array operations and array slices instead of loops, as well as built-in parallel instructions (`where`, `forall` etc.): it is faster

Fortran 95, 2003 and 2008 has several looping/array assignment constructions that have been optimised for speed in multi-processor parallel environments. Never use loops to initialise arrays, and avoid using them to calculate array components. Whenever possible, *reverse the order of indices* in nested loops, e.g. first looping should be over the columns, and then over the rows. Nested loops may have huge speed overhead! Use `FORALL`, `WHERE` and similar new "parallelized" Fortran constructions. Below is a little test conducted on an average amd64 system using GNU Fortran (`-O3 -funroll-loops -fforce-addr`, timing is by Linux `time`).

Test 1: Multiple nested loops, execution time = 0m12.488s

```
use BASE_UTILS
use BASE_RANDOM
implicit none
integer, parameter :: n=1000, a=100,b=100,c=100
integer :: nn, i,j,k
real :: random_r
real, dimension(a,b,c) :: M   ! The above header part is the same in all tests

call random_seed_init

MATRLOOP: do nn=1,n
  random_r = rand_r4()
  do i=1,a                                        ! Multiple nested loops
    do j=1,b
      do k=1,c
        M(i,j,k) = random_r
      end do
    end do
  end do
end do MATRLOOP
```

Test 2: Direct array assignment, execution time = 0m1.046s

```
! header the same as above...
call random_seed_init

MATRLOOP: do nn=1,n
  random_r = rand_r4()
  M=random_r                                      ! Direct array assignment
end do MATRLOOP
```

Test 3: forall instruction, execution time = 0m1.042s

```
! header the same as above...
call random_seed_init

MATRLOOP: do nn=1,n
  random_r = rand_r4()
  forall (i=1:a, j=1:b, k=1:c) M(i,j,k) = random_r  ! Parallel instruction
end do MATRLOOP
```

Test 4: Reverse order of nested loops (cols then rows), execution time = 0m1.046s

```
! header the same as above...
call random_seed_init

MATRLOOP: do nn=1,n
  random_r = RAND_R4()
  do i=1,a
    do j=1,b
      do k=1,c
        M(k,j,i) = random_r                         ! Order of looping is reversed
      end do
    end do
  end do
end do MATRLOOP
```

Multiple nested loops with the most "natural and intuitive" indices order (rows then cols) had a *really huge* execution speed overhead [6], more than *ten times* slower than the other methods (compare 12.5s and 1.0s!). The code is also more concise and easier to read. The same tests with Oracle Solaris Fortran (`f95`) turning on aggressive optimization and automatic loop parallelization (`-fast -autopar -depend=yes`) run much faster, but the speed differences still remained quite impressive (first test execution time = 0m0.010s, all other = 0m0.006s). So compiler-side aggressive CPU optimisation does work, although the tricks remain very useful.

Note that newer versions of Fortran compilers can become smart enough to adjust the order of looping in the machine code. Nonetheless it is better to write "optimised" code using the above tricks that works fast just everywhere.

13 Object-oriented programming and modelling

13.1 General principles

Modern Fortran (2003 and 2008 standards) allows coding in a true object-oriented style, although does not require it. Object oriented style allows to define user's abstractions that mimic real world objects, isolate extra complexity of the objects and create extensions of objects.

Object oriented programming is based on the following principles:

Abstraction: defining and abstracting common features of objects and functions.

Modularity and hiding irrelevant information: An object is written and treated separately from other objects. Details about internal functioning of the object are effectively hidden, what is important is the *interface* of the object, i.e. how it interacts with the external world. This reduces complexity.

Encapsulation: combining components of the object to create a new object.

Inheritance: components of objects (both data and functions) can be inherited across objects, e.g. properties the "genome" object inherited by a more general object "the individual."

Polymorphism: the provision of a single interface to objects of different types.

At the most basic level the programmer defines both the **data structure** (user's type) as well as the types of **operations** (subroutines and functions) that are linked with and are be applied to the data structure.

Important
Object-oriented features of Fortran are described in recent Fortran books, e.g.: **Brainerd, W.S. (2009)**. *Guide to Fortran 2003 Programming,* Springer (Chapter 12). **Chapman, S.J. (2007)** *Fortran 95/2003 for Scientists and Engineers,* 3rd ed., McGraw-Hill (Chapter 16). **Chivers, I. & Sleightholme, J. (2012)** *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77,* Springer (Chapter 26). Short introduction can also be found on the internet, e.g. **Leai, M.** *Object-oriented programming in Fortran 2003* (PGI: www.pgroup.com). Part 1: Code Reusability; Part 2: Data Polymorphism.

[6] This is because allocation of arrays in the computer memory goes in an "index-reverse" order in Fortran, see http://www.fortran90.org/src/best-practices.html#multidimensional-arrays

13.2 Implementation of objects

It is the most convenient and natural to define a single object or closely related objects within the same Fortran module. Note also that components of an object or derived type are referred using the percent symbol `%`, e.g. `fish%sex` refers to a component `sex` of the object `fish`. Both derived type data components and functions are referred in this way. Derived type data objects can be combined into arrays as "normal" data. For example, the `sex` component of the *i*-th element of the array of derived type `fish` is referred as `fish(i)%sex`. Note that derived types can also include arrays, so `predator%prey(j)` can be *j*-th element of the `prey` array component of the object `predator`. If we use an array of derived type that includes a three-dimensional array component, it could be something like `fish(i)%position(x,y,z)`.

Data structure (user-defined type) is defined in Fortran using the keywords: `type ... end type`. An object can also include subroutines and/or functions. For example, the following object `INDIVIDUAL_GENOME` includes a data structure consisting of a single character string `str` and two subroutines that define its behaviour. The first subroutine has the internal name `init_genome_random` but is referenced outside of the object as `init_genome` (i.e. `init_genome` is a part of the object's interface').

```
module THE_GENOME                               ! The module defines GENOME object
....

type, public :: INDIVIDUAL_GENOME               ! It is defined here
  character(len=len(GA_TARGET)) :: str          !   we have user (derived) type
  contains                                      !   ... and type-bound procedures.
    procedure, public :: init_genome => init_genome_random
    procedure, public :: mutate => mutate_genome
end type INDIVIDUAL_GENOME

private :: init_genome_random, mutate_genome ! Internal names are "private," so
                                                !  the outside procedures can refer
                                                !  the object subroutines by their
                                                !  outer "interface" names set on
contains                                        !  the left of "=>"

  subroutine init_genome_random(this)           ! The subroutine is almost as usual
    class(INDIVIDUAL_GENOME) :: this            ! Note the use of the CLASS keyword
    ........
  end subroutine init_genome_random
....
end module THE_GENOME
```

Note that the subroutine part of the object `init_genome_random` must have an item of the type definition (`this`) as its first argument. However, we must define it as `class()` rather than `type()`. With `class`, the subroutine will work not only with this specific `type`, but also with any of its extension (i.e. it is a polymorphic type).

We may then define an additional, more general, object extending the `INDIVIDUAL_GENOME`. In this case, we use the word `extends` in the new type definition (see code below). This says that the components of the `INDIVIDUAL_GENOME` are also included into (the new object `INDIVIDUAL_NEURO_ARCH` (i.e. `INDIVIDUAL_NEURO_ARCH` inherits the `INDIVIDUAL_GENOME` components).

```
module THE_NEUROBIO
.....
type, public, extends(INDIVIDUAL_GENOME) :: INDIVIDUAL_NEURO_ARCH
  integer :: bundles
  contains
    procedure, public :: init_neuro => init_neurobio_random
end type INDIVIDUAL_NEURO_ARCH

private :: init_neurobio_random

contains

  subroutine init_neurobio_random(this)
    class(INDIVIDUAL_NEURO_ARCH) :: this
```

```
    ........
  end subroutine init_neurobio_random

end module THE_NEUROBIO
```

In this way, it is easy to create new objects inheriting properties of other objects, for example, create several layers ranging from the **genome** through the **neurobiological architecture** and up to the **individual fish** and further to a **population** of fish.

However, the above is just the *definition* of an object. To use the object, we must *instantiate* it, i.e. create its specific instance and set the values. This is analogous to having a specific data type, e.g. `integer`. We cannot use "just an integer," we need (1) to create a specific variable (variable is also an object though trivial!) of the type *integer* (e.g. `integer ::Var_A`) and (2) to asign a specific value to it (`Var_A =1`).

For example, the following creates two instance arrays of the type `INDIVIDUAL_FISH`. Both arrays are one-dimensional and have `POPSIZE` elements. So we now have two fish populations, `generation_one` and `generation_two`. But instead of being arrays of simple values they are arrays of complex data structures potentially consisting of many different data types and arrays:

```
type(INDIVIDUAL_FISH), dimension(POPSIZE) :: generation_one
type(INDIVIDUAL_FISH), dimension(POPSIZE) :: generation_two
```

We can now assign concrete values to each of the previously defined components of `generation_one` array, e.g.

```
generation_one(i)%sex = "male"     ! assign values to individual components
generation_one(i)%alive = .true.   ! of the object instance
```

We can also use the subroutines and type-bound functions that we have defined within the object definitions to do specific manipulations on the object and its components:

```
subroutine population_init()
....

  do i = 1, POPSIZE
    call generation_one(i)%init()
  end do

end subroutine population_init
```

TODO — more text later

14 Final Notes

There are a few other modules. I will write similar documentation for them too… Hope it is soon. There is still much to to.

The manual is generated with AsciiDoc markup processor. Later, an auto-generation of docs from the model code is planned (not first priority though).

15 Index