Error handling in Fortran 2003[1]

Koen Poppe, Ronald Cools, Bart Vandewoestyne[2]

Department of Computer Science, KU Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{Koen.Poppe,Ronald.Cools,Bart.Vandewoestyne}@cs.kuleuven.be

Abstract

Although the Fortran programming language is evolving steadily, it still lacks a framework for error handling — not to be confused with floating point exceptions. Therefore, the commonly used techniques for handling errors did not change much since the early days and do not benefit from the new features of Fortran 2003. After discussing some historical approaches, a Fortran 2003 framework for error handling is presented. This framework also proved to be valuable in the context of unit testing and the design-by-contract (DBC) paradigm.

Categories D.1.5 [Object-oriented Programming]; D.2.4 [Software/Program Verification]: Validation, Programming by contract; D.2.5 [Testing and Debugging]: Error handling and recovery; D.3.3 [Language Constructs and Features]: Frameworks.

Keywords Error handling, error codes, ifail, Fortran 2003, Unit testing, design-by-contract (DBC).

1. Introduction

Once the concept of error handling is put forward, it is obvious that for numerical software a wide range of tests are required. While regular software will mostly compare integers or objects, numerical software must be able to compare floating point numbers in a robust fashion. Exact equality for those numbers is too strong a condition and so absolute or relative errors will be used as criteria. Note that, as arrays are important in Fortran, it should be straightforward to compare these in an element-wise fashion using the same criteria.

Error handling is not the only case in which these comparisons will be useful. In unit testing and in adopting the design-by-contract paradigm [10], very similar functionality is required. Therefore, although this publication focusses on error handling, the described package can and has been used in these different contexts.

By revisiting well known approaches for error handling in Section 2, we pinpoint some key ingredients for an error handling framework. Section 3 summarises those features and introduces a conceptual error handling framework. Before concluding, the main result is presented in Section 4: a Fortran 2003 package for error handling whose error handling operations are summarised and documented in the Appendix A.

2. Well known approaches

2.1 Integer error codes

In absence of an error handling mechanism built into the Fortran standard, unexpected behaviour was historically propagated by integer codes. Conventionally, zero indicates normal behaviour while non-zero indicates some kind of unexpected situation. The codes themselves are passed on using arguments commonly named ifail[3] or inform.

By careful design and a centralised approach to these error codes within a certain scope, this provides a suitable approach to deal with runtime errors. It has been applied in numerous packages: LINPACK [4], EISPACK [5, 12], LAPACK [1], MPI [9], PVM [6], NAG Fortran Library [14], CUBPACK [3], amongst others, and in several intrinsic routines of the Fortran standard [7] itself: **stat** for allocations, **iostat** for input/output, ...

However, integer error codes do not scale very well. A centralised module with all the codes and their descriptions is needed in order to avoid duplicates. This is reasonable for small packages, but difficult to maintain in growing software projects. Also, because the description of the event resides in this centralised module, it is hard to ensure that the right error code is returned in the right situation and that the description, given when the error is reported, of the situation is up to date.

[1] This is an updated version of the article published as [11].

[2] This paper presents research results of the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State Science Policy Office. The scientific responsibility rests with its authors.
The authors acknowledge the support for this project by the Bijzonder Onderzoeksfonds of the Katholieke Universiteit Leuven.

[3] Because of the FORTRAN 66 implicit typing rules, a variable named fail would implicitly be **real**, therefore the additional i was added. This way ifail is implicitly defined to be an **integer**.

Besides scaling issues, this approach also has drawbacks due to the use of integers: the compiler is by no means able to check whether the indicated error code actually exists, so typing mistakes might be hard to track down. Also, because there is no verification, collisions between error codes can occur. This might lead to bugs that are difficult to detect, as shown in Listing 1.

Also, each occurrence of the error code requires some explanation because, unless you are familiar with the software, this single number does not tell the whole story. The same is true for the end-users of the program, therefore, some packages that follow this approach also provide a method to translate each situation into a human readable string. Note however that a single integer cannot conceal the full extent of the encountered unexpected situation. For example, if a program requests a username that can only consist of alphanumeric characters (i.e., a-z, A-Z and 0-9), an integer error code approach could signal invalid characters. But would it not be better to also give a list of the disallowed characters that have been detected? Similarly, if the username must be unique, it could be helpful to return a slightly modified username that is known to be unique so that the user does not have to try all obvious choices?

Lastly, it is common practice to define the ifail argument as **intent(in out)**. The input value does allow the calling routine to decide how the called routine should deal with unexpected situations[4]. We feel however, that this makes programming more complex because the meaning of this variable changes during the runtime of a procedure: from way of handling to actual error code. The calling routine must be careful not to mistake those two states of the ifail variable and the called routine must always reflects this change.

Besides the above drawbacks, Fortran developers have been applying this paradigm over and over, as mentioned before, and so does the Fortran standard. Also, the convention that zero/nonzero signifies success/failure should not be changed overnight.

2.2 Error codes with message

The XERROR package [8] provides additional functionality on top of the error code approach. It acknowledges that the code itself, or even fixed messages, are not sufficient in certain situations and that actual (numeric) values should be presented. It introduces several levels of severeness of errors ranging from a warning, recoverable error up to fatal errors. The original version written in FORTRAN 77, has recently been ported to Fortran 90 [2], but still, the earlier mentioned drawbacks of the use of error codes are still unsolved.

[4] Common terminology in this context is *soft silent*: do not report, continue execution; *soft noisy*: continue the program, but report the error; or *hard*: report and stop the program.

Listing 1. Combining user-defined codes with processor dependent codes might lead to collisions and wrongly identified unexpected conditions: if the **allocate**(...) on line 43 would return a *processor dependent* **stat**=2, the resulting ifail will be mistaken for the allocation of a negative number of elements (line 17).

```
1  program collision
2    implicit none
3    integer, dimension(:), allocatable :: array
4    integer, parameter :: memory_bound = 1000000
5    integer :: nb_elements, ifail
6
7    read(unit=*,fmt=*) nb_elements
8    call my_allocation( array, nb_elements, ifail )
9
10   select case( ifail )
11     case( 0 )
12       ! NOP, all went fine
13     case( 81001 )
14       write(unit=*,fmt="(A,I0,2A)") &
15         "Allocation of over ",memory_bound," elements is",&
16         "not allowed due to fixed memory bound"
17     case( 81002 )
18       write(unit=*,fmt=*) &
19         "Allocation of a negative number of ", &
20         "elements is not allowed in this context"
21     case default
22       ! ifail = 81000+stat argument of allocate( ... )
23       write(unit=*,fmt=*) &
24         "Allocation failed, stat=", (ifail-81000)
25   end select
26
27 contains
28   ! Allocation subroutine
29   subroutine my_allocation( array, nb_elements, ifail )
30     integer, dimension(:), allocatable, intent(out) :: array
31     integer, intent(in) :: nb_elements
32     integer, intent(out) :: ifail
33     integer :: allocation_stat
34
35     if( nb_elements > memory_bound ) then
36       ifail = 81001
37     else if( nb_elements < 0 ) then
38       ifail = 81002
39     else
40       allocate( array(nb_elements), stat=allocation_stat )
41       if( allocation_stat /= 0 ) then
42         ! allocation_stat is processor dependent
43         ifail = 81000+allocation_stat
44       else
45         ifail = 0 ! All went fine
46       end if
47     end if
48   end subroutine my_allocation
49 end program collision
```

2.3 Error type with message

In the NAG Fortran 90 Library [13], a *derived-type*-based approach to error handling is presented. The usual error code is encapsulated in a derived type which, besides an error code, also contains a message describing the unexpected situation in much more detail. It also contains additional options to regulate the behaviour and the output and discriminates between *warnings*, *failures* or *fatal* situations. This approach provides better feedback using the descriptive message, but still has drawbacks.

First of all, the variable is still **intent**(**in out**) and thus requires an explicit initialisation because the value, on input, determines the behaviour in case of an unexpected situation. The manual also states that the user *must* test if the results are valid but this cannot be enforced. As is the case with error codes, the programmer can still ignore the unexpected result indication, voluntary or involuntary. It would be helpful if this was no longer possible.

3. Our ideal error handling

When developing a new framework for error handling, it should be developed in an evolutionary sense: existing code should require only minimal changes to be able to work with the framework. It can even reduce the code size by encapsulating frequently occurring checks. Ideally, the language itself could support this, but a collection of wrappers for certain routines, as in Listing 2, also makes it more convenient to use the framework.

It should be possible to gradually introduce more advanced error handling features without significant code overhead, especially avoiding explicit initialisation. Another reason for the evolutionary approach is the compatibility with error mechanisms that are built into the Fortran standard (**stat**, **iostat**, ...). Therefore, the developed system must be almost interchangeable with the well known paradigms, for example, detect abnormal behaviour using ifail/=0 as it is usually done.

When discussing error handling, an issue that is frequently mentioned is the performance penalty. This seems paradoxical because errors, meaning the programming model in which errors abort the execution of the current subroutine, are often used in situations where defensive programming, i.e., checking everything before starting, is not advisable because of the performance impact that would incur. However, the argument still stands: when encountering an unexpected situation, the overhead should only include the storage of all information necessary to characterise the situation. This way, if the calling routine decides that an alternative approach is possible, no time is wasted.

Printing a lot of error messages on the screen is one way to determine where and when an unexpected situation was encountered and how it propagated through the program. However, as programs get more complex, messages may be printed that are not relevant to the failure of the program as a whole (see Listing 3). Therefore, it seems better to postpone the output to the moment when there is no-one who might take care of the situation. At that moment, the full trace of errors should be printed, giving a clear top-down or bottom-up overview of what happened. Note that this is in line with the store-only principle of the previous argument.

Clear responsibilities is another important requirement. It seems reasonable that, no matter what situation is encountered, only the calling routine should decide what to do with it. Take the example from Listing 3: if a memory allocation fails, then maybe another (slower?) algorithm that uses less memory can be triggered. Taking the previous requirement into account, the choice is binary: the caller complies with taking responsibility for the situations that might happen, or the called routine is on its own and can do whatever it pleases. Note that this avoids the distinction in how to handle unexpected situations (cf. *soft noisy*, ...) because the caller decides afterwards and can do so *on an error to error basis* (in contrast to, for example, speci-

Listing 2. Illustration of how the verification of the allocation of an array can be encapsulated in order to reduce the amount of frequently reoccurring code. Adding this to an error handling framework should simplify the adoption and might even reduce the code size of the users program.

```
1  module error_handling_common_wrappers
2    use error_handling_error
3    implicit none
4    interface allocate
5      module procedure allocate_integer_2_rank1
6      ! ... similar for other ranks, kinds and types
7    end interface
8    ! ...
9  contains
10   subroutine allocate_integer_2_rank1( array, shape, ifail )
11     integer(kind=selected_int_kind(2)), &
12     dimension(:), allocatable, intent(out) :: array
13     integer, intent( in ) :: shape
14     type(error), intent( out ), optional :: ifail
15
16     integer :: allocation_stat
17     allocate(array(shape),stat=allocation_stat)
18     if( allocation_stat /= 0 ) then
19       call create_error( ifail, &
20         allocation_error( allocation_stat, (/ shape /) ) )
21     end if
22   end subroutine allocate_integer_2_rank1
23   ! ... similar for other ranks, kinds and types
24 end module error_handling_common_wrappers
```

Listing 3. Printing warnings in allocate_workspace (in case the allocation at line 1 would fail) must be avoided: it only confuses the end user because we do have a backup plan in place for that specific situation.

```
1  call allocate_workspace( N*N, ifail )
2  if ( ifail == 0 ) then
3    ! Fast algorithm with O(N^2) memory complexity
4  else
5    call discard_error( ifail )
6    call allocate_workspace( N, ifail )
7    if ( ifail == 0 ) then
8      ! Slower algorithm with O(N) memory complexity
9    else
10     ! Not possible to solve the problem for given N
11   end if
12 end if
```

fying *soft noisy* for *all* errors). Consider for example an optimisation routine which cannot ensure all optimality conditions for the given problem setting: if the calling routine depends on that optimum, this might be fatal although it could have been reported as a warning.

Furthermore, there should be ways to deal with groups of unexpected situations besides the fine-grained approach of individual errors. Consider for example grouping all illegal argument errors, all operating system signals or processor dependent errors. This naturally leads to a hierarchical structure of errors which allows end users to deal with classes of unexpected behaviours or individual errors if suitable.

As mentioned at the end of Section 2.3, the methodology should be complete in the sense that no errors can be ignored. The two obvious options are to report it or to propagate it. However, some errors might be harmless and thus require no further action. In order to ensure this completeness, we propose an explicit method to discard the error. If not discarded, the framework must still be able to report the unexpected situation and abort the program, indicating this programming mistake.

Similar to the discarding of errors, it should be possible for the compiler to check if a certain special situation exists. Literal values, like error codes, are clearly not an option and should be avoided in any case. Also, the errors should somehow be characterised by something that provides the programmers or readers an indication about what they represent.

When encountering an unexpected event, the response should not be to automatically return from the current routine. This is often misused to create **goto**-like structures. Instead, the execution continues and might allow some cleaning up, or a second attempt, before gracefully returning to the calling routine.

To summarise, at least the following generic operations should be available to the program developer:

- **Creation** – When an unexpected situation occurs, an error is created. This includes storing all parameters that are needed to uniquely define the situation. This operation is often referred to as *throw*, *raise*, *signal*, *trigger*, ...
- **Transfer** – If a called procedure terminated with an error, this can be taken into account and another method can be used. However, some errors cannot be recovered on the current level. Therefore, the responsibility for the error can be transferred to the calling procedure. If the calling procedure is not prepared to take the responsibility, the error must be reported. Instead of transfer, some approaches use *rethrow*, *pass*, *propagate*, ... for this.
- **Chaining** – When it is not meaningful to propagate a certain error, another type of error can be created. This enables the translation of errors and the combination of several errors into a single *umbrella*-error. It is important to keep the information about the original error as the reason for the new one in order to be able to track down what caused it. This can be useful for example to encapsulate errors from within a package to one single *internal error* type.
- **Reporting** – There must be a way to inform the end-user of the program about what went wrong. Usually, a detailed report should be shown on the screen including all relevant information. Note that it is only at this point that the stored parameters of the special situation are translated into a meaningful human-readable message intended for the end-user.
- **Discarding** – If a certain error is not problematic, an explicit discarding is needed to ensure that the error is not forgotten somehow. In case an error is not discarded explicitly, i.e., when it was wrongly/involuntary ignored, it must still be able to report itself.
- **Check for errors** – When a called routine returns the control, there should be a straightforward way to check whether an error occurred or not, preferably also in the historically familiar fashion: **if**(ifail/=0).... It must also be possible to check what type of error occurred, both on individual error type and a more coarse grained level.

4. Proposed package

The following paragraphs detail the package that has been developed with all the concerns of Section 3 taken into account. It is based on the Fortran 2003 standard [7]. Although some compilers support all the features used in this package, several do not. Therefore, some features of the Fortran 2003 standard can be avoided with specific compiler flags. More details can be found in the 'ReadMe' and Makefile accompanying the package.

Listing 4. Usage of the error derived type from the viewpoint of the main program that calls the linear system solver with diagonal matrix from Listing 5.

```
use linear_system_solver
real(kind=wp), dimension(10) :: diag, b, x
type(error) :: ifail
! ... Initialisation
x = solve( diag, b, ifail )
if ( ifail == 0 ) then
  write(unit=*,fmt="(A)") "Solution returned in x:"
  write(unit=*,fmt=*) x
else
  call report_error( ifail )
end if
! ... Initialisation
x = solve( diag, b )
! in absence of the ifail optional argument, the solve method
! decides what to do in case of unexpected situations
write(unit=*,fmt="(A)") "Solution returned in x:"
write(unit=*,fmt=*) x
```

4.1 Basic usage and operations

The basic entity to represent errors is the derived type **type**(**error**) which is essentially a polymorphic pointer to a **class**(**error_info**) that stores the details of the circumstances in which the error occurred. Making a distinction between the error and the associated information simplifies the use: it encapsulates much of the technical control structures concerning the actual pointers, while still providing enough flexibility to represent different types of unexpected situations.

When applying a routine that is equipped with this type of error handling, such as the example in Listing 4, one can indicate that one wants the responsibility of unexpected behaviour by providing a **type**(**error**) argument. Verifying that all went fine can be done similarly as with the integer code approach as can be seen in Listing 4. This makes it easy to migrate existing codes using the usual convention. The rudimentary solver of Listing 5 illustrates how special situations can be reported through this optional output argument by calling the **create_error** subroutine. This takes the information about the unexpected situation as the second argument and will take care of the fact that the ifail argument might not be present. Note that, as mentioned before, this does not end the function, therefore explicit **return** statements are added.

If an error cannot be handled in a certain routine, it can in turn pass it on to its caller by using the **transfer_error** routine as illustrated in Listing 6. Note that an assignment is not equivalent to this, because an assignment cannot have any side effect and thus duplicates the error and the responsibility. This is the

Listing 5. Linear system solver with diagonal matrix: use of the error type from solvers viewpoint. Includes the declaration of the **type**(**illconditioned_error**) with its custom formatting function.

```
module linear_system_solver
  use error_handling
  implicit none
  type, extends(error_info) :: illconditioned_error
    character(:), allocatable :: matrix_name
    real(kind=wp) :: lambda_min, lambda_max
  contains
    procedure :: write_to => illconditioned_write_to
  end type illconditioned_error

contains
  ! Solve a system of equations with diagonal matrix
  function solve( diag, b, ifail ) result( x )
    real(kind=wp), dimension(:), intent(in) :: diag
    real(kind=wp), dimension(size(diag)), intent(in) :: b
    type(error), intent( out ), optional :: ifail

    real(kind=wp), dimension(size(diag)) :: x

    real(kind=wp) :: lambda_max, lambda_min

    ! Is the problem ill conditioned?
    lambda_max = maxval(abs(diag))
    lambda_min = minval(abs(diag))
    if ( lambda_min < lambda_max*epsilon(1.0_wp) ) then
      call create_error(ifail, illconditioned_error( &
        "diag", lambda_min, lambda_max) )
      return
    end if

    ! 'Solve' the diagonal system of equations
    x = b / diag

  end function solve

  subroutine illconditioned_write_to( info, unit, prefix, suffix )
    class(illconditioned_error), intent(in) :: info
    integer, intent(in) :: unit
    character(len=*), intent(in) :: prefix, suffix

    if( info%lambda_min == 0.0_wp ) then
      write(unit=unit,fmt="(5A)") prefix, &
        "Matrix ", info%matrix_name, " is singular", suffix
    else
      write(unit=unit,fmt="(7A,ES10.3,A)") prefix, &
        "Matrix ", info%matrix_name, " is ill conditioned:", &
        " cond(", info%matrix_name, ")=", &
        info%lambda_max/info%lambda_min, suffix
    end if
  end subroutine illconditioned_write_to
end module linear_system_solver
```

purpose of the **transfer_error** routine: it transfers the responsibility from one **type**(**error**) object to another.

In user interface routines of packages, it is often not desirable to transfer all internal errors to the user. Instead, a more user friendly error can be given or several types of errors can be grouped into a single one. However, to allow expert users to detect what went wrong internally, the original internal error should be added to new one somehow. This is the purpose of the optional reason argument of **create_error** in Listing 6.

At some point in the responsibility chain, the unexpected situation cannot be handled by robust programming and must be reported. To this account, the **report_error** is available that takes the information of the error and transforms it into a human readable output. Evidently, the format of this output depends on the information that is stored in the specific type that is derived from **type**(**error_info**).

If a certain unexpected situation is recoverable and an alternative approach is possible, an error might be discarded with the **discard_error** routine. As argued before, this must be done explicitly. If not, the framework will still report it at the end of the scope of the error. This is accomplished by tracking whether or not an error has been reported. When the **type**(**error**) is finalised but never reported, the package will report it for you, causing the program to abort.

4.2 Predefined errors

In order to minimise the overhead of common unexpected situations, several wrapper functions are included in the package. This includes a routine similar to Listing 2 for the allocation of arrays.

There is also the **type**(**error_code_error**) which stores a certain error code (can be used while transitioning from the error code approach) and a **type**(**message_error**) which contains only a description of the situation.

The package also adds design-by-contract [10] features. In this paradigm, all preconditions and postconditions are explicitly added to the code as a kind of contract. This way, when testing the software, violations of these contracts can be detected. Because these checks might incur a performance penalty, there is a way to disable them in the release version of performance critical production code.

4.3 User defined errors

One of the features of the package lies in the ability to define custom error information types. The derived type approach provides several benefits, most noticeably, the ability to detect groups of errors based on the type they were derived from, as illustrated in Listing 6.

Defining a **type**(**error_info**) derived type, i.e., a type that stores all relevant information about a certain situation, is straightforward. As an example, the declaration of the **illconditioned_error** is shown in Listing 5. Only when a certain error needs to be reported, the stored information is transformed into a human readable message. This is accomplished by overriding the write_to member function.

Listing 6. The hierarchical structure of the **error_info** derived types allows the program to discriminate between individual as well as groups of errors.

```
subroutine library_interface( ifail )
  type(error), intent(out) :: ifail
  type(error) :: inform
  call internal_routine( inform )
  select type( inform%info )
    type is( no_error )
      ! equivalent with (inform==0)
    class is( internal_error )
      ! encapsulate internal errors in a message_error
      call create_error( ifail, message_error( &
        "Error in the <PackageName>"), reason=inform )
    class default
      ! other types of errors -> propagate to caller
      call transfer_error( inform, ifail )
  end select
end subroutine library_interface
```

4.4 Advanced numeric features

It is not uncommon to work with arrays with a rank higher than one, therefore specifically designed routines are added to show the difference between arrays (including matrices) in a fashion that is both easy to inspect (i.e., differences are highlighted) and compact (i.e., only relevant slices of the array are shown).

With numerical routines in mind, relative and absolute errors can be used as a criterion for these differences as illustrated in Listing 7 and 8. Note that the framework is nor limited to scalar variables nor to real data: complex data types are also supported besides integers and logicals. Also, during compilation, all available kinds are detected and suitable routines are generated for each of them. It is thus possible, for example, to use assert_relerr for rank-2 arrays of complex variables with extended precision (if the compiler supports this precision).

5. Conclusion

We have presented a framework for error handling that benefits from a number of features introduced in Fortran 2003 and provides several benefits over the historical techniques using error codes, both in terms of ease-of-use as robustness.

Listing 7. Parts of the illustration on the use of the framework for design by contract programming or unit testing. The output of these tests is shown in Listing 8.

```
program dbc_example
  use error_handling
  implicit none
  integer :: n, fact
  double precision, dimension(7,2) :: A,B

  do n=-3,20,3
    write(unit=*,fmt="(A,I2,A,I10)",advance="no") &
        "n = ", n, " -> n! = ", factorial(n)
  end do

  ! ... A = reference, B = approximation
  call assert_relerr( A,B, epsrel=1.0d-12 )

contains
  function factorial( n ) result( fact )
    integer, intent(in) :: n
    integer :: fact
    integer :: i
    ! Preconditions
    call precondition_ge( n,0, "Factorial undefined for n<0" )
    ! Main
    fact = 1
    do i=2,n
      fact = fact*i
    end do
    ! Postconditions
    if( n == 0 ) then
      call postcondition_eq( fact,1, "Definition 0!" )
    else
      call postcondition_ge( fact,1, "Factorial >=1" )
    end if
  end function factorial
end program dbc_example
```

Listing 8. Output of a program similar to Listing 7. Observe how rows 1, 4 and 5 are excluded from the output and that the differences in digits are indicated.

```
n = -3 -> n! =
*** Error:
   Run-time check: a >= b (Factorial undefined for n<0)
   a =          -3
   b =           0
n =  0 -> n! =           1
n =  3 -> n! =           6
n =  6 -> n! =         720
n =  9 -> n! =      362880
n = 12 -> n! =   479001600
n = 15 -> n! = 2004310016
n = 18 -> n! =
*** Error:
   Run-time check: a >= b (Factorial >=1)
   a =           -898433024
   b =                    1
*** Error:
   Run-time check: |(a-b)/a| <=  1.0E-12
   4 of the 14 elements differs:
    T T
    F T
    T F
    T T
    T T
    F T
    T F
(equal rows suppressed)
a(2,:)= 3.163822112410288E-02   6.704883136950639E-01
b(2,:)= 3.100000000000000E-02   6.704883136950639E-01
        ---^^^^^^^^^^^^-^^^----
relerr= 2.017247182132713E-02   0.000000000000000E+00
------
a(3,:)= 1.078708132514429E-01   8.007757414134162E-01
b(3,:)= 1.078708132514429E-01   8.007750000000000E-01
                               --------^^^^^^^^^^----
relerr= 0.000000000000000E+00   9.258689766348766E-07
------
(equal rows suppressed)
a(6,:)= 8.385771850213821E-01   1.969016923561929E-01
b(6,:)= 8.385771850213821E+12   1.969016923561929E-01
        ----------------^^^
relerr= 9.999999999999000E+12   0.000000000000000E+00
------
a(7,:)= 9.844135616404036E-01   1.649799989920529E-01
b(7,:)= 9.844135616404036E-01  -3.141592653589793E+00
                               ^^-^-^^^-^^^^^^^^^^-^-^
relerr= 0.000000000000000E+00   2.004226374580790E+01
```

A. Quick reference

As shown in the examples in this part, variables containing the errors are of the type **type**(**error**) while the information about the error is derived from **type**(**error_info**). Using [argument] for optional arguments, the following interfaces are available:

- **create_error**([ifail], info, [reason], [method]), where ifail is the variable storing the error[5], info contains the information about the error. The reason argument can be used to chain another error as origin for the current, while method may be used to provide the name of routine in which the error occurred.
- **transfer_error**(inform, [ifail]) transfers the error from inform to the optional ifail argument. If the ifail argument is not present, the error will be reported.
- **report_error**(ifail, [fatal]) that outputs the given error to the unit determined by report_error_unit() and can be set through set_report_error_unit(**unit**),
- **discard_error**(ifail) indicates that the given error has been taken care of and can be silenced safely.
- **allocate**(array, shape, [ifail]) a wrapper for allocating arrays of all available types and up to rank 3 with a given shape.

In the context of unit testing, the following assert statements are available, for all types and available kinds up to arrays with rank 2.

- assert(condition, [OPTIONS])

[5] This ifail argument must be written in the code, but the argument itself might also be an optional argument and thus it can still be that it is not present at run-time. Not-present optional arguments can be passed on following the Fortran standard.

- assert_OP(a,b, [OPTIONS]), where OP is replaced by one of the following: **eq**, **ne**, **lt**, **le**, **gt** or **ge**
- assert_abserr(a,b, epsabs, [OPTIONS]), uses abserr(a,b)
- assert_relerr(a,b, epsrel, [OPTIONS]), uses relerr(a,b)

where [OPTIONS] is an abbreviation for one or more of the following: [comment], [ifail], [statement] a textual representation of the condition, [a_name] and [b_name] that give a descriptive name to the expressions given as arguments a and b, [filename], [line] and [**fmt**] that determines the output format of the actual values.

For the design-by-contract methodology, the above methods are available through the interfaces where assert_... is replaced by precondition_..., postcondition_... or check_.... The only difference with the above routines is the fact that the actual error that might be given differs and that the tests can be disabled using

- dbc_setup([**all**], [precondition], [postcondition], [check]).

The logical arguments enable or disable specific tests which are enabled by default. Note that the **all** argument has a lower priority than the individual flags, so enabling all tests except for preconditions can be done for example using dbc_setup(**all**=.true., precondition=.false.).

When writing unit tests, one can use the same subroutines but this time with a unittest_... prefix instead of assert_.... These will not abort the execution but simply keep track of how many of the tests succeeded and how many failed. With unittest_reset([suite_name]), the counts can be reset and a name for the test-suite configured, while unittest_report will report an overview of the results so far. Typical unit tests will thus be similar to the following:

```
1 call unittest_reset( "suite_name" )
2 call unittest_...( ... )
3 ! ... more tests
4 call unittest_...( ... )
5 call unittest_report()
```

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

[2] J. Burkardt. XERROR error message handler. URL `http://people.sc.fsu.edu/~jburkardt/f_src/xerror/xerror.html`.

[3] R. Cools and A. Haegemans. Algorithm 824: CUBPACK: a package for automatic cubature; framework description. *ACM Trans. Math. Softw.*, 29(3):287–296, 2003. ISSN 0098-3500.

[4] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979. ISBN 0-89871-172-X (paperback).

[5] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines — EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science, Editors: G. Goos and J. Hartmanis*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1977. ISBN 0-387-08254-9, 3-540-08254-9.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0262571080.

[7] ISO/IEC JTC1/SC22. Information technology - Programming languages - Fortran - Part 1: Base Language. Technical report, ANSI, October 2003.

[8] R. E. Jones and D. K. Kahaner. XERROR, the SLATEC error-handling package. *Software: Practice and Experience*, 13(3):251–257, 1983. ISSN 1097-024X. doi: 10.1002/spe.4380130305.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*, september 2009.

[10] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, 2 edition, 1997.

[11] K. Poppe, R. Cools, and B. Vandewoestyne. Error handling in Fortran 2003. *Fortran Forum*, 31(2):7–19, August 2012. doi: 10.1145/2338786.2338787.

[12] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines — EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science, Editors: G. Goos and J. Hartmanis*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1976. ISBN 0-387-06710-8.

[13] The Numerical Algorithms Group (NAG). *The NAG Fortran 90 Library (fl90) Manual*. Oxford, United Kingdom, 4 edition, 2000.

[14] The Numerical Algorithms Group (NAG). *The NAG Fortran Library Manual*. Oxford, United Kingdom, 22 edition, 2009.