Debugging Guide

The following is a compilation of common, trivial but "invisible" errors that may have catastrophic results. The erroneous action is marked with red.

Common Errors: Precision, Expressions

- Mixed expressions that produce non desired results are used. Example (both a, b are real):

  a = 4.0
  b = 1/2 + a**2

  Here b will be equal to 16.0 after execution. The correct expression is:
  b = 1.0/2.0 + a**2

- Tolerance criteria cannot be met because of finite precision. Example:

  Suppose we are using Newton's method to solve numerically an equation. If we write:

  DO WHILE (Err1 /= 0.D0)

  we will never converge because the error cannot be exactly equal to zero. The right way to do this is:

  DO WHILE (dabs(Err1) >= Tol)

  were Tol is a predefined small number indicating Tolerance, e.g. Tol = 1.D-8.

- A series of expressions containing many variables is evaluated. The variables are incorrectly updated. Example:

  We know the position of a fly in 3D space x, y, z at time t and we have an algorithm that evaluates its position at the next timestep t + dt. The algorithm says:

  $x_{next} = 1.08*x_{current} + 0.02*y_{current}$
  $y_{next} = y_{current} + 0.14*x_{current}$
  $z_{next} = 1.01*z_{current} + 0.01* x_{current}**2 + 0.04*y_{current}$

  This will be manifested as follows:

  DO
  $x_{next} = 1.08*x_{current} + 0.02*y_{current}$
  $y_{next} = y_{current} + 0.14*x_{current}$
  $z_{next} = 1.01*z_{current} + 0.01* x_{current}**2 + 0.04*y_{current}$
  $x_{current} = x_{next}$
  $y_{current} = y_{next}$
  $z_{current} = z_{next}$
  ENDDO

The following structure is incorrect and will yield erroneous results because x and y are updated prematurely:

```
DO
  x_current = 1.08*x_current + 0.02*y_current
  y_current = y_current + 0.14*x_current
  z_current = 1.01*z_current + 0.01* x_current**2 + 0.04*y_current
ENDDO
```

- Signs are incorrect. In many cases the whole code is correct except from a single sign and after many hours of debugging we realize how simple the problem was.

Common Errors: Loops, Branches

- A DO loop is initialized with a counter, but inside the loop another counter is used. Example:

```
DO i = 1,10
   x(j) = y(j)**2
ENDDO
```

We should have used I instead of j inside the loop.

- The same counter is used in nested loops. Nested loops can never have the same counters, e.g. the following structure is invalid:

```
DO i = 1,10
   DO i = 1,10
      x(i,i) = y(i)**2
   ENDDO
ENDDO
```

- A quantity is calculated through a recursive expression, but the variable used is not initialized first. Example:

Suppose we want to calculate the sum $\sum_{i=1}^{10} \frac{1}{i^2}$. This is automatically a DO loop:

```
DO i = 1,10
   Sum = Sum + (1/i**2)
ENNDO
```

Now if this is the first time variable Sum is used we may not have problem, because Fortran give the value Sum = 0.D0 upon initialization of the variable. But if Sum has previously been used and has a value, say, Sum = -198.9381D0, then the final value calculated in our loop will be erroreous. To be sure that you get correct results always initialize your variables:

```
Sum = 0.D0
```

```
DO i = 1,10
  Sum = Sum + (1/i**2)
ENNDO
```

Common Errors: Subroutines, Functions

- A function is used with a CALL statement. A function is called by its name, e.g. y = func1(a,b).

- A subroutine is defined with a set of arguments but in the CALL statement used to execute the subroutine, different arguments are used. Example:

```
PROGRAM main
IMPLICIT NONE
REAL temperature, distance, x, y, z

x = 19.0
y = 2.0
z = 7.8
distance = sqrt(x**2 + y**2 + z**2)
CALL tempcalc(temperature,distance)

! Commands follow …

END
```

```
SUBROUTINE tempcalc(temperature,x,y,z)
IMPLICIT NONE
REAL temperature, distance, x, y, z

distance = sqrt(x**2 + y**2 + z**2)

! Commands follow …

RETURN
END
```

The programmer here created tempcalc(temperature,x,y,z) first, but when writing the main program he forgot that the arguments of tempcalc involve x, y and z. He thus wrote CALL tempcalc(temperature,distance) to trigger the subroutine which is invalid. He should have written:

```
CALL tempcalc(temperature,x,y,z)
```

in the main program.

- A subroutine updates a variable but the result is not returned to the calling program. Be sure to include all the desired input and output variables in the definition of a subroutine. Example: we want to calculate temperatures and return them in the main program but we forget to include variable temperatures in the definition:

  ```
  SUBROUTINE tempcalc(x,y,z)
  IMPLICIT NONE
  REAL temperatures, distance, x, y, z

  ! Commands follow …

  temperatures = …

  ! Commands follow …

  RETURN
  END
  ```

  Even if we have the command that calculates the temperatures, the results are not returned to the main program unless the subroutine definition is as follows:

  ```
  SUBROUTINE tempcalc(temperatures,x,y,z)
  ```

  Always remember that the variables used in a subroutine or function are “local”, dummy variables (unless COMMON blocks are used of course).

- Arguments in a subroutine or function create memory site conflicts upon execution of the subroutine. Consider the following call statement…

  ```
  CALL updateaux(indx(1),indx,prqu,n,n0)
  ```

  …for subroutine:

  ```
  SUBROUTINE updateaux(i,indx,prqu,n,n0)
  ! Definitions…

  indx(1) = 51

  ! Commands follow …
  ```

  Now suppose that at some point in the execution of the subroutine, the value indx(1) changes, as in our example indx(1)=51. This change will inevitably affect variable i of the subroutine as well, because of the way the call statement was constructed. That is immediately after execution of the assignement command: indx(1)=51, variable i will have the value 51 even though we did not have any command such as i = 51. The way to avoid this is to rephrase the CALL command as follows:

  ```
  k=indx(1)
  call updateaux(k,indx,prqu,n,n0)
  ```

Common Errors: Arrays

- An array is defined with fixed subscript ranges and in the program there is reference to an element "outside" of the array bounds, e.g. we define x(20,20) and we have a command x(21,21) = 3.2. This produces "Array bounds exceeded" errors upon execution; however some compilers may not detect the error.

- Fixed size array is defined with different subscript ranges in a program and a subroutine. Example:

```
PROGRAM main
IMPLICIT NONE
REAL temperatures(10), height

height = 10.0
CALL tempcalc(temperatures, height)

! Commands follow …

END
```

```
SUBROUTINE tempcalc(temperatures, height)
IMPLICIT NONE
REAL temperatures(20), height

! Commands calculating temperatures follow …
! height is not assigned any new value (we have no 'height = …' commands)

RETURN
END
```

This may have very strange effects in your program, depending on the compiler, for example variable height may have different values before and after execution of tempcalc, even though the subroutine did not assign any new value to height.