MET3220C
Computational Statistics

Programming: Debugging

Dr. Mark Bourassa

Where are Things Most Likely To Go Wrong?

- 1st place to check: reading data
  - Test: As the data is read, write the data to the screen.
  - Look at the contents of the data file.
  - Make sure the data written to the screen is consistent with the data file.
- 2nd place: passing data to or from subroutines
  - Reread the lecture notes on typical subroutine errors
  - Test: in the subroutine, write the input data to the screen
    - Is it consistent with the similar data in the program calling the subroutine?
  - Test 2: If data are output from the subroutine, then print the data in the subroutine and in the main program (after calling the subroutine).
    - Are they consistent?

Floating Point Exceptions

- You have an equation that is trying to calculate something that doesn't make sense.
  - Examples:
    - Dividing by zero,
    - Log of zero or less,
    - Squareroot of zero or less
    - Dividing a very large number by a very small number
  - Test: put some PRINTs in the code to determine where the problem occurs.
    - The program will run until the floating point error is encountered.
    - The print statements can be used to write suspect variables to the screen

The code compiles, but the result is bad

- Suspect #1: non-initialization of variables used in sums.
  - You have allowed the computer to set the initial value of the variable – it could be anything.
  - Test: look for initialization in code.
    - If need be, use grep to find all occurrence of the variable
- Suspect #2: Used integer math. Dividing an integer by an integer will result in rounding to the whole number closest to zero.
  - Examples: 5 / 2 = 2; 2 / 5 = 0;   1 / (n – 2) = 0 if $n \neq 3$
  - Solution: Convert integers to real values, e.g. 1.0 / REAL(n – 2).
- Suspect #3: Garbage in – garbage out.
  - You messed up earlier in the code, and are using bad values in the equation.
  - Test: print the values of the variables to the screen

The code compiles, but the result is bad

- Incorrect brackets:
  - Very common currently in this class
  - Example 1:
    - Delta = (index) * ((sum_x_sqd) – ((sum_x)**2))
    - There is NO NEED for most of these brackets. Math rules apply, with the exception of implicit multiplication.
    - The above equation can be simplified to Delta = index * (sum_x_sqd - sum_x**2)
    - Which is inconsistent with

$$\Delta = n\left(\sum_{i}^{n} x_i^2\right) - \left(\sum_{i}^{n} x_i\right)^2$$

    - And can be written as Delta = index * sum_x_sqd - sum_x**2

FORMATTING ERRORS

- You are using a formatted write, and the output is a bunch of starts
  - Example: *******
  - This means that the data cannot fit in the specified format because the value attempting to be printed is too large.
    - Example: trying to squeeze 4556 into an I3
    - Example: trying to squeeze 100.7 in F6.3
    - Advice: check the size with an unformatted print or write.
- You are printing 0.0 for a small but non-zero number
  - Example 0.00005 appears as 0.000
  - This means that you are not specifying enough decimal places to the right of the decimal.
    - Example: 0.00004 formatted as F5.4 results in .0000
    - Example: 0.4 formatted as I3 results in   0
    - Advice: check the size with an unformatted print or write.

Is There a Bug in This Code?

```
DO i = 13, 1272 ! Begin reading actual data
  READ(7,'(I4,1X,I2,T9,F6.2,T19,F6.2,T29,F6.2)')year, month, min,
   max, rai
  IF (ierror /= 0) EXIT ! Exit loop at end of file or read error
  ! missing data check
  IF (min /= -99.99 .AND. max /= -99.99 .AND. rai /= -99.99) THEN
    index = index + 1 ! Ugly time array
    time(index) = REAL((year - 1900) * 12 + month)
    tmin(time(index)) = min  ! Associate min/max/rain with index
    tmax(time(index)) = max
    rain(time(index)) = rai
  ENDIF
ENDDO
Blah blah blah
CALL bestfit(tmin, time, index, slope, sig_slope, y_int, sig_yint)
```

- Or is the problem in the subroutine?

Is There a Bug in This Code?

```
SUBROUTINE bestfit( y, x, index, slope, sig_slope,
   y_int, sig_yint )
IMPLICIT NONE
INTEGER :: i, x(index), index
REAL :: y(index), y_int, slope
REAL :: sig_yint, sig_slope
REAL :: sum_x, sum_y, sum_x_sqd, sum_xy
REAL :: DELTA, sig_y, sum_ymxb
sum_x = 0
sum_y = 0
sum_x_sqd = 0
sum_xy = 0
DO i = 1, index !
  sum_x = sum_x + x(i)                 ! Sum date points
  sum_y = sum_y + y(x(i))              ! Sum temp/rain data
  sum_x_sqd = sum_x_sqd + x(i)**2      ! Sum sqr of dates
  sum_xy = sum_xy + x(i) * y(x(i))     ! Sum product
ENDDO
```

The Answers

- Both the codes are OK by themselves.
- However they are inconsistent in the arguments
- The array lengths in the subroutine will be shorter than the array lengths in the main program.
  - Index is the number of points MINUS the missing values.
- Solutions:
  - 1) Fill the data arrays (tmin, tmax, and rain) with only good data (no change yet), but like the time array don't create gaps for missing data.
    - Pass in only the values from with good data.
    - For example, tmin(1:index)
  - 2) Make the time array similar to the data arrays, and have the subroutine filter out bad data
    - The down side is that your subroutine has to recognize bad data, which might have different indicators in different data sets.

Segmentation Faults

- Segmentation faults occur when trying to write past the end of an array or something similar.
- Example 1: tmin(index) = 10.
  - When index is outside the array bounds
- Example 2: passing to much data to an array or subroutine.
  - Call my_cool_function(3.14159)
  - When my_cool_function expects an integer

Messing Up with Array Indices

- Consider declaration of variables. This process sets up a block of memory to be used to hold the information associated with these variables.
- Example:

  INTEGER :: n_bins, qscat_flag, n_good_data, index_spd, max_num_spd, status

  REAL, DIMENSION(700) :: pdf_obs, pdf_gaussian, pdf_log_normal

  REAL, DIMENSION(700) :: cdf_obs, cdf_gaussian, cdf_log_normal

  REAL, ALLOCATABLE, DIMENSION(:) :: qscat_spd_array

  REAL :: bin_width, qscat_spd, sum_qscat_spd, sum_qscat_spd_sqd, small, standev_qscat_spd, standev_log_spd, max_spd, min_spd

  REAL :: log_spd, sum_log_spd, sum_log_spd_sqd, PI, bin_center, mean_qscat_spd, mean_log_spd
- Space for allocatable arrays is usually later in memory.

Messing Up with Array Indices

- If we write to an array location that is outside the array bounds, then we are modifying other variables! YIKES!
- What would the following do?
  - pdf_obs(n_good_data) = qscat_spd
- The value of n_good_data should be between 1 and about 800,000.
- The index for pdf_obs should range from 1 to 700
- Moral: be more careful with array indices. Using the wrong index, outside the bounds of the array, is kind of like taking a shot gun to the program's memory.
- How do you test for this problem?
  - Compile with a –C (upper case 'C')
  - This checks each time an array index is used to verify that it is within bounds.
  - It slows down the code, so if the code is going to be reused, it is practical to recompile without the this compiler flag.
- f90 –flags gives a list of compiler options.

Array Wizardry in FORTRAN90

- The WHERE command is a combination of a DO loop and an IF.
  - It performs array operations, but only on (or using) elements for array elements that meet the condition.
  - Note: the logical test should be applied to an array!
- Syntax 1:
  - WHERE (*logical test*) *array operation*
- Syntax 2:
  - WHERE (logical test)

    array operation(s)

    ENDWHERE
- Syntax 3:
  - WHERE (logical test)

    array operation(s)

    ELSEWHERE

    array operations(s)

    ENDWHERE