ATSC 212

Earth & Atmospheric Science
Intro Computing Lab

Last Lab - Conclusion

Roland Stull

rstull –at- eos.ubc.ca

Topics

1. Computational pitfalls with Floating-point binary math
2. Synthesis of topics and methods.
3. Final exam preparation.
4. Summary and Review.
   - Recap of good programming style.
   - Your future programming activities.
5. Goals Achieved.

Reminder to do the online instructor evaluations (as notified by the Faculty of Science),

and the online TA evaluation (via our course Vista webpage/Assessments).

1. Binary Calculations: Pitfalls with Floating-Point Numbers

(work will be done on the blackboard, based on the readings.)

- representation of floating point numbers in binary – accuracy, precision, & range
- errors: round-off; truncation
- pitfalls: conditional testing with floats; small difference between large floats; be aware of the discrete number line
- tips: add smallest numbers first; reduce number of math operations; use existing math libraries (Num. Recip.)

Binary Representation of Floating-point (real) numbers

- scientific notation: mantissa and exponent
- example: decimal: 8.453 x 10^4
  - mantissa = 8.453 , exponent = 4
  - mantissa = 0.8453, exponent = 5
  - the mantissa has 4 digit precision
- example: binary for a 32-bit real number
  - mantissa has roughly 24 bits (23 bits plus a sign bit)
  - exponent has roughly 8 bits (but varies with CPU type)
  - mmmmmmmmmmmmmmmmmmmmmmmmeeeeeeee
  - 11000011100011010010101110111100
  - +10000111000110100101011 x $2^{10111100}$

Range and Precision

- example: binary for a 32-bit real number
  - mantissa has 24 bits , exponent has 8 bits
  - +mmmmmmmmmmmmmmmmmmmmmmm x $2^{eeeeeeee}$
  - But varies slightly with CPU type
- Max value of exponent is 128
  - Max **range** is 2^{128} = 10^{38} on some CPUs
  - Max **range** is 2^{125} = 10^{37} on other CPUs
- A 24-bit mantissa can represent about $\pm 2^{23}$ different numbers, or 6 to 7 significant decimal digits of **precision**.
- Any number with more digits will be rounded off to 6 or 7 digits of precision.
- This is **round-off error.**

Pitfalls: Conditional tests using real numbers (floats)

Write a Fortran program to be as follows. Then compile and run.

```
program floater         !test floating point numbers
  implicit none         !enforce strong typing
  real :: x = 0.0       !initialize a floating-point variable

  write(*,*) "Welcome to Floater”
  do
    x = x + 0.1               !increment x
    if (x .eq. 3.0) exit      !stop looping when x = 3
    if (x .gt. 10.0) exit     !stop looping when x > 10
    write(*,"(F4.1)") x       !display the value of x
  enddo

endprogram floater
```

Pitfalls: Conditional tests using real numbers (floats)

Did the program do what you expected? Why?

Let's run some diagnostic tests to understand why. A useful debugging trick is to add more write statements to get more info about the values of variables as the program runs.

So AFTER the existing write statement for x, add the following write statement that is not formatted. Then save, recompile and run.

```
write(*,*) x          !display the value of x
```

1) Based on these diagnostics, what was the problem?

2) Can you suggest ways to fix it?

Increasing the Precision of Real numbers.

Perhaps if the binary number were more precise, it might successfully execute the program.

In Fortran, a real number with extra precision is called a "**Double Precision**" (dp) number. If often uses 64 bits (8 bytes) to hold the floating point number, which has a **precision** of about 14 or 15 decimal digits,
and a **range** of about $\pm 10^{307}$ or $\pm 10^{308}$.

But it varies from CPU to CPU.

Increasing the Precision of Real numbers.

```
!Useful intrinsic functions related to precision:
  precision(x)    !returns the precision of real variable x
  range(x)        !returns the range of real variable x
  selected_real_kind(p=pr, r=ra) !returns the “kind number”
     of the lowest-precision real number that satisfies your
     desired precision pr and range ra.
```

The “kind number” is an integer code that specifies the precision of real numbers. Often ,but not always, it is a count of the number of 8-bit bytes used to represent the number. On some machines, kind number 4 indicates a normal, single-precision number. Kind number 8 indicates double precision. But depends on the CPU.

Try it. Modify your code:

```
program floater      !test floating point numbers
  implicit none      !enforce strong typing
  integer, parameter :: sp = selected_real_kind(p=precision(0.0))
  integer, parameter :: dp = selected_real_kind(p=precision(0.0)+1)
  real :: x = 0.0    ! initialize a floating-point variable

  write(*,*) "Welcome to Floater"

  write(*,*) "Single precision real kind is sp =", sp
  write(*,*) "Double precision real kind is dp =", dp
  write(*,*) "sp precision, range =", precision(0.0), range(0.0)
  write(*,*) "dp precision, range =", precision(0.0_dp), range(0.0_dp)

  do
    x = x + 0.1              !increment x
    if (x .eq. 3.0) exit     !stop looping when x = 3
    if (x .gt. 10.0) exit    !stop looping when x > 10
    write(*,"(F4.1)")  x     !display the value of x
    write(*,*) x             !display x atwith full precision
  enddo

endprogram floater
```

…then compile and run.

Increasing the Precision of Real Numbers.

Real-number constants in Fortran CAN be written in normal decimal form (e.g., 3.14159). Or they can be written in scientific notation, such as 2.84E6 (which represents 2.84x10^{6}). Single precision real numbers are the default, so you don’t need to specify their kind.

Double-precision numbers MUST utilize the kind specifier, which had to have been previous declared as a Parameter.

```
integer, parameter :: dp = selected_real_kind(p=precision(0.0)+1)

real(dp) :: x      !shows how to declare a double-precision variable
…
x = 3.1415926535_dp     !shows how to write a double-prec. constant
```

Try Increasing the Precision of the Real numbers in your code

Continue trying different solutions to get your program to stop when encountering the conditional test if (x .eq. 3.0), until you get one that works.

What do you suggest we should try?

Are there alternatives that don't involve double precision?

NOTE: These issues are NOT limited to Fortran, but apply to every language that uses real numbers.

The Number Line

See the paper by Hayes on "The higher arithmetic."

How are decimal numbers distributed along the number line when represented by binary: (a) integers, (b) floats (discuss at blackboard)

Numbers too small to be represented cause **underflow**, and are usually truncated to zero.

Numbers too large cause **overflow**, and cause the program to STOP with an error message (if you are lucky).

Math operations that cause Bit shifting reduces precision

To add or subtract real numbers, the exponents of the two numbers must match.

For example in decimal, to add $2.46\text{x}10^5 + 3.15\text{x}10^3$. For the smaller number, we must first shift the digits in the mantissa 2 places to the right and increase the exponent by 2 in order to make the exponents match before doing the addition. Assume the mantissa has precision of 3 digits.

$$\begin{array}{rcl} 2.46\text{x}10^5 & & 2.46\text{x}10^5 \\ \underline{+\ 3.15\text{x}10^3} & \text{is the same as} & \underline{+\ \ 0.03\text{x}10^5} \\ & & =\ \ 2.49\text{x}10^5 \end{array}$$

But we lost precision in the small number before adding.

Similar bit shifting and exponent incrementing is done for binary additions.

Tips to minimize round-off errors:

1) add small numbers together first before adding larger ones.
2) reduce the number of arithmetic operations if possible.
3) avoid small differences between 2 large numbers.

Truncation Errors

See paper by Press et al p 30, and Chapman p649-650. **Truncation error** is a characteristic of a program or algorithm, while **round-off error** is a characteristic of your computer hardware.

Truncation error refers to truncating the number of terms in an infinite series approximation to a complex function, such as

$e^x = 1 + x/1! + x^2/2! + x^3/3! + ...$

All complex functions (sin, cos, exp, ...) in computer languages are approximated by series of simpler operations (+, -, *, /), but they can't retain the infinite number of terms in the series. Hence, the series is truncated to a fine number of terms.

Tips to minimize truncation errors:

1) use libraries of functions that have been created and tested by others. (e.g., the ones published in "Numerical Recipes")

2) in a series, add the smallest terms first.

2. Synthesis of Concepts

- For each programming language we studied, for which situations is it the best language to use? (i/e; advantages of one vs. another)
- For which situations would unix/linux be a better operating system to use than Windows or MacOSX?
- What are the programming tools my grad students use?
- What additional programming topics would you like to see taught in this course in future years?

Utility of what you learned:

Computer Skills used by ATSC co-op Students:

As undergrad co-op student	D.B.	M.W.
EC	unix/linux SQL (MS-SQL) Fortran windows visual basic excel	unix/linux Fortran windows visual basic excel bash scripting TecPlot (graphics)
UBC	unix/linux html SQL Fortran C Perl php python	unix/linux html bash scripting php Vis5D (graphics)

3. Final Exam Preparation

Date, Time, Place. Open book. Open computer. About 10 questions.

Readings (revisited):

- Course Pack section on Computer-language Evolution:
  - Hayes, B., 2006: The semicolon wars. American Scientist, 94, 299-303.
  - Levenz, E. 2009: History of programming languages. (a timeline chart)
- Course Pack section on Good Programming Practices:
  - Section from: B.W. Kernighan and R. Pike, 1999: The Practice of Programming. Addison Wesley)
    - Epilogue
    - Appendix - Collected Rules
    - Chapter 5 - Debugging (focus on main topics and methods, not on any one programming language)
  - Section from: S.J. Chapman, 2007: Fortran 95/2003 for Scientists & Engineers, 3rd Ed. McGraw Hill. Only pages 82-89, which cover top-down design and flowcharting.
- Course Pack on Binary Calculations and Pitfalls:
  - Press, Teukolsky, Vetterline, Flannery, 2007: Numerical Recipes in C, 2nd Ed. only pages 28 - 31 in Section 1.3 Error, Accuracy, and Stability, which covers binary numbers, numerical accuracy, and numerical errors.
  - S.J. Chapman, 1998: Fortran 90/95 for Scientists & Engineers, 1st Ed. McGraw Hill. pages 4-11 and p646-660, which covers round-off errors, truncation errors, and other in numerical-calculation errors.
  - B. Hayes, 2009: The higher arithmetic. American Scientist, 97, 364-368.

4. Summary. You learned:

- Good programming practices
- Common tricks and pitfalls in scientific programming
- Hands-on experience with
  - Tools: KompoZer, VI, Emacs
  - Compiled Languages: FORTRAN, C, Perl
  - Operating System: UNIX/Linux
  - Database: MySQL
  - Web authoring: html

Some Thoughts

- Free vs. Commercial software …
  - You get what you pay for.
  - Advanced tools: debuggers, profilers, editors, etc.
- Needs of a science manager:
  - A badge of your skill is NOT compact code.
  - I unhappily throw away $ every time a new programmer decides to write new code from scratch instead of building upon code written by my previous programmers.
  - Therefore, make your code EASY to understand.
- Don’t write obscure code to enable job security.

Programming Realities (again)

- You are a physical scientist or engineer.
- You are programming for yourself.
- Your goal is to solve a physical problem, or to simulate the workings of nature.
- Once the program is successfully written, debugged, and tested, you will use it only once, or a few times.
- The most expensive part of this activity is
  - You (writing the program)
  - Not the computer (running the program)
- Once the program works, you will change it before you run it again.

Tips (again)

- Think of the whole scientific problem before you start writing code. (Think before you write.)
- Program from the top down.
- Test your code incrementally, as you write it.
- Use Version control.
- Document each line as you write it, using in-line comments. Include physical units.
- Add full-line comments as header paragraphs
  - Explain what aspects of physics your code describes.
  - Cite main eqs. from journal papers or books.

… more Tips (again)

- Write clear, simple, logical code.
- Use strong typing (declare all variables).
- Don’t use cryptic variable names.
- Indicate the physical units via in-line comments with your type declarations, or in the variable name itself.
- Initialize variables during declaration.
- Test intermediate & extreme values against known outcomes or hand calculations.

Namely, follow the Rules in the Appendix of Kernighan & Pike.

…and even more Tips

- Always type in the closing parenthesis at the same time you type the opening parenthesis.
- Don’t implement changes operationally just before you leave on vacation, or on Friday just before the weekend.
- Pick a language appropriate for your task: Excel, Matlab, Java, C, FORTRAN, Perl, Python,…
- Use “canned” libraries or algorithms where possible, such as is in “*Numerical Recipes*”.

5. Goals Achieved

ATSC 212 Earth & Atmos. Intro. Computing Lab

- Exposed you to different programming languages used in science & engineering.
- Increased your chances of getting a job or being accepted into grad school.
- Allowed you to list on your resume that you' ve written some code in these various languages.
- Gave you the confidence (and exposure to resources) that you need to learn more about these languages on your own.

The End