Outline

- Announcements:
  - HW I key online this afternoon
  - HW II due Friday
  - Sign up to discuss projects
- Debugging
- Testing for correctness

Development Process

1. Design: What will the program (or modification) do? How will it work?
2. Specification—formal statements of what code will do
4. Implementation: write the code
5. Build: Get it to compile and run
   a) Debug I: find and fix syntax errors
   b) Debug II: find and fix semantic errors (testing)
6. Improve performance through tuning or re-design

Old-Fashioned Debugging

- The point of debugging is to find your errors
- Simplest technique is checkpointing
  - Place an output statements around calls to subroutines
    • printf("Entering subroutine A")
    • A();
    • printf("Completed subroutine B")
  - If your program crashes in A, you won’t see the second line
- Work into subroutines, bracketing sections of code with outputs until you find where the error occurs.
Old-fashioned Debugging

- Enhancements to checkpointing:
  - Print values of suspect variables
  - Could involve saving arrays to files
    - Useful to have routines to write specific data types
    - If graphics are available, plot the data

- Checkpointing is nice because it works on any system that can run your code
- But, requires lots of compiles as you zero in on bug.
- WARNING: Finding the line where the program crashes is not enough, you need to know why!
  - The problem could result from a previous statement
  - In this case, figure out where the variables on the offending line are set, and work backwards

Old-Fashioned Debugging

Middle-age debugging

- UNIX standard debugging program is db (gdb on Linux)
- gdb allows you to watch your program run
  - Set breakpoint--position in code, execution will stop when reached
  - Step through program line-by-line
  - Examine value of variables
- Special compiler settings are required (-g)
  - db needs to know correspondence between object code and source code
Modern Debugging

- IDEs like VizStudio have graphical debuggers
  - On some systems, this is just a GUI for db

Validation

- Assume your code will compile and run
- THIS DOESN'T MEAN YOU'RE DONE
- You need to verify that your code is solving the the right problem

Formal Specification

- $I \land P \Rightarrow O$
  - This says what a program should do, but says nothing about how it will get done
  - No details of $P$
- We now have $P$ (we think)
  - We can test that $I \land P \Rightarrow O$ for some inputs and outputs
Test Cases

- Test cases are an important part of scientific computing
- Typically, these are simple problems for which the answer is known
  - Ideally, an analytic solution
- Examine output--systems like MATLAB are useful

Test Cases

- Test cases are also useful for sharing and extending code
  - Provide a suite of test cases to new users
    - Tests that code runs on their system
    - Helps them learn to run your code
  - Make sure your extensions didn’t break anything!

Specification of model problem

- English: Given inputs, RAD1d finds an approximate solution for PDE:

\[ \frac{\partial C}{\partial t} = u \frac{\partial C}{\partial x} + \frac{\partial}{\partial x} \left( k \frac{\partial C}{\partial x} \right) + r(C, x, t) \]

  at time T
  - Sol’n will be more accurate as dx & dt ->0
Test Cases for Model Problem

\[
\frac{\partial C}{\partial t} = u \frac{\partial C}{\partial x} + \frac{\partial}{\partial x} \left( D \frac{\partial C}{\partial x} \right) + r(C, x, t)
\]

- Advection only:
  - If \( u \) is constant, then regain initial conditions at \( t = \frac{L}{u} \)
- Diffusion only, constant:
  - Complicated analytical solution, but know that solution should tend to mean(\( C \))
- Reaction only
  - Solution should agree with ODE: \( \frac{dC}{dt} = \text{reaction} \)
    (each grid point represents a unique initial condition)

Flaw of Test Cases

- Test cases are simple and usually won't test all parts of the code
- Pick small, but hard cases to test logic of program
  - Small enough that you can reason through yourself
- Test subroutines individually, then test program
- Key of testing:
  - Look at your output!

Next 2 weeks

- Wed: Debugging tools
- Fri: Version control & working in groups
- Mon & Wed: Profiling and tuning (lecture and lab)
- Fri: Project presentations (5 min)
  - General description of problem (diagram?)
  - Mention tools used & what you learned
- Other ideas?