12.010 Computational Methods of Scientific Programming

Lecturers
Thomas A Herring
Chris Hill

Today’s class:

- Basic content of course and class expectations
- Overview of languages
- Overview of program development

Class meetings:

- Lectures are 2 hours (break in middle). Latter parts of class will sometimes be used for in-class tutorials
- Lectures will be held here in 54-322
- Specific textbooks and web resources will be recommended at time of each section

Class expectations

- Homework will be set once every couple of weeks (basically one homework per section in the course)
- You may collaborate on the homework, but your submissions should be your own work.
- There is no final exam, but there will be a final project due at the end of semester.
- 5/6 grading on homework, 1/6 grading on final project.
- Web page for course contains homework dates . The 2010 web site contains examples from previous years.

Introduction

- Languages to be covered:
  - Fortran, Matlab, Mathematica, C, C++, Python and graphics and advanced (parallel and GPU computing) topics
- Specific versions:
  - ANSI Fortran77 with Fortran 90 differences
  - Matlab Release 2011a
  - ANSI C and C++
  - Mathematica Version 8.0.1

  (ANSI: American National Standards Institute web.ansi.org)

Aims of this course

- Cover both program design and syntax of languages
- At the end of the course: Write one language and read all languages
- Understand the differences between the languages so that appropriate one is chosen for when needed later
- We will not cover all aspects of each language but we will try to give you a working knowledge of the languages.

Basic distribution of classes

- 2 introductory lectures discussing general problem solving
- 5 lectures on Fortran 77 and Fortran 90/95
- 4 lectures on C and C++
- 2 lectures on Mathematica
- 4 lectures on Matlab
- 2 lectures on Python
- 4 lectures on parallel computing and advanced topics including graphics card computations
- 2 lectures on graphics and numerical methods

Language concepts Fortran, C, C++

- Compiled languages i.e., source code is created with an editor as a plain text file.
- Programs then needs to be “compiled” (converted from source code to machine instructions with relative addresses and undefined external routines still needed).
- External routines are those needed to do such operations as read disk files, write to the screen, read the keyboard strokes etc.)
- The compiled routines (called object modules) need then to be linked or loaded.

Fortran, C, C++ cont.

- Linking creates an executable with relative addresses resolved and external routine loaded from the system and user libraries.
- The executable can then, in most cases, be run on any machine with the same architecture.
- Compiling and linking can be done in one user step.

Fortran, C, C++ cont.

- *Advantages:* Programs are fast and can be run on many machines that don't need to have Fortran or C++ loaded.
- (There are exceptions to this depending on linking options. Safest is "static linking" in which case system libraries are loaded into program. Dynamic linking expects to find specific versions of system routines in specific locations.)

MatLab

- Interactive language with some automatic compiling (into MatLab pseudocode) of user subroutines (M-files).
- User programs can be developed as scripts
- Speed of modern processors makes this type of approach reasonable.
- Graphics and Graphical user interfaces (GUI) are built into program (for Fortran, C, and C++ graphics must be done through external or user written routines).

Matlab continued

- *Advantages:* Since interactive, user can debug on the fly, results available immediately (i.e., values of specific variables can be viewed at any time. In Fortran, C++ code must be changed to output values or a debugger program used). Integrated environment which can guide program development and syntax
- *Disadvantages*: Code can only be exported to users with the same version of Matlab but can often be used on different platforms depending on whether platform specific options are used). Code can be often converted to C and compiled (license needed)

Mathematica

- Interactive symbolic manipulation program with built in computation and graphics.
- This type of program is often used to derive algorithms for MatLab, Fortran and C++ but can also be used to generate results.
- Work can be organized into “work-books” that can be extended as a project progresses
- Program has options to export code and formulas in word processing languages

Mathematica

- *Advantages:* Symbolic manipulation so that it yields formulas rather than numerical results. Analysis of equations gives insights into characteristics of problems.
- Can represent numerical values with arbitrary accuracy
- *Disadvantages:* Codes need version of Mathematica. Viewing notebooks also requires some parts of Mathematica be installed.

Python

- Python is an interpreted language that shares many features with C and Matlab. These types of languages are often called "scripting" languages
- It has common approach to "quick" solutions to problems and has a very large community of developers (open source)
- No variables need be declared and often program development can be fast in this language.
- Program are created with a text editor.
- The name comes from Monty Python's Flying Circus.

Parallel computing

- One of the methods available to carrying out large computations. Basic idea is to break problem into smaller parts that do not depend directly on each other and can be evaluated simultaneously on separate computers.
- Linux based PCs make this approach relatively inexpensive.
- Not quite “off-the-shelf” yet but progressing rapidly (e.g., later in course we will look at parallel Matlab).

General approach to any computational problem:

- Statement of problem: The clearer this can be done, the easier the development and implementation
- Solution Algorithm: Exactly how will the problem be solved.
- Implementation: Breaking the algorithm into manageable pieces that can be coded into the language of choice, and putting all the pieces together to solve the problem.
- Verification: Checking that the implementation solves the original problem. Often this is the most difficult step because you don't know what the “correct" answer is (reason for program in the first place).

Example case

- How many numbers need to be stored to save a symmetric NxN matrix in lower diagonal form? (This is our statement of problem: Note this may be a small piece of another larger problem).
- Solution algorithm: In lower diagonal form, the number of values needed go as: 1x1=1, 2x2=3, 3x3=6 ...
- To increase the dimension of the matrix from (N-1)x(N-1) to NxN requires N new values. Therefore our algorithm is the sum of integers from 1 to N.
- *So how do we solve it?*

Possible solutions

- Call the number of values we need num_elements. How do we compute this value?

(a) Simplest

num_elements = 1 + 2 + 3 + 4 + 5 + ... + N

- – Coded specifically for specific values of N. Not very flexible; could be many options depending on number of values of N possible.

(b) Intermediate solution

- Does not require much thought, takes advantage of looping ability of most languages:
  - **Fortran77**:

```
num_elements = 0
do i = 1, N
  num_elements = num_elements + i
end do
```

  - **Matlab**:

```
num_elements = 0;
for i = 1:N ;
  num_elements = num_elements + i ;
end ;
```

(c) Final solution

- (c) Requires some thought about the sequence (also looking up the sum of series in a book like Abramowitch and Stegun (eds), Handbook of Mathematical Functions, Dover, New York,).

  num_elements = N*(N+1)/2

- All of the above are examples of algorithms that can be used. The implementation will depend specifically on system used.

Verification of algorithm

- What can fail in the above algorithms. (To know all the possible failure modes requires knowledge of how computers work). Some examples of failures are:
- For (a):What happens if specific value of N needed is not coded?

Need some exception handling.

- For (b): This algorithm is fairly robust. (See (c) for some possible problems).

When N is large execution will be slow compared to (c).

Algorithm (c)

- This is most common algorithm for this type of problem, and it has many potential failure modes. For example:
- (c.1) What if N is less than zero?

Still returns an answer but not what would be expected. (What happens in (b) if N is negative?).

Algorithm (c) 02

- (c.2) In which order will the multiplication and division be done.

For all values of N, either N or N+1 will be even and therefore N(N+1) will always be even but what if the division is done first? Algorithm will work half of the time.*

If division is done first and N+1 is odd, the algorithm will return a result but it will not be correct. This is a bug. For verification, it means the algorithm works sometimes but not always.)

Algorithm (c) 03

- (c.3) What if N is very large?

What is the maximum value N(N+1) can have? There are maximum values that integers can be represented in a computer and we may overflow. What happens then? Can we code this to handle large values of N?*

- We will return later to algorithm development when languages are discussed in more detail

Input/Output (IO)

- Important part of any program since this is where human interaction occurs.
- Generally programs start with some type of input to be supplied by human, and end with output of results that humans need to interpret.
- IO comes in many forms.
  - Maybe simple reading of keyboard for user reply or could be interrupt driven reading from serial or modem port (real-time data transfer).
  - We will cover file and screen IO but not real-time IO operations (the latter are very platform dependent).

Input/Output (IO) 02

- IO operations typically involve human interaction and therefore the code has to be very robust since the actions of humans are unpredictable.
- The role of IO is important to consider when designing an algorithm (Think of programs you have used where either the input or the output is not satisfactory.
- IO can be problematic especially when an expected input never arrives.

Verification

- Basically: How do you know that the results of your program are correct?
- The overall program is so large and complex, that human checking of the the results is usually not possible.
- Remember there are very rarely answers in the back of the book.
- **There are no definite rules for this operation**.

Verification 02.

- By breaking the program into small modules, each of which can be checked, the sum of the parts is likely to be correct but not always.
- Note: getting a program to compile and run is only the first (small) step in programming.
- Problems can be that the program only works some of the time (sound familiar), or it may not work on all platforms.
- **The critical issue to realize all possible cases that might occur.**

Verification 03

- This takes experience and the more you know about how computers work, the more likely you are to realize the exceptions.
- When humans are involved or you are working with instrument data streams you need to be particularly careful.

Next Lecture

- In the next class we:
  - Examine basic components of computers
  - Number representation
  - Methods of approaching the solution to different classes of computer problems.