15. 780.20 Session 15

a. Overview of Session 15

As we wind down to the last week of the quarter, we’ll tie off some loose ends and hit some topics that have been omitted so far. Here’s the gameplan:

- 3D plots using gnuplot;
- using the gdb debugger;
- taking a look at optimization;
- using the Intel C++ compiler;
- trying out a profiler.

b. Optimization

In an ideal world, optimization of a computer code would be transparent to the user: the compiler would do it for you. In practice, different compilers for the same language on the same machine can provide very different performances. That is why people still pay big bucks for fortran compilers rather than use g77. In the Linux C++ world, we have g++ and then commercial compilers. The Intel C++ compiler, which we’ll explore in this session, has a very strong benchmarking record, although the latest version of g++ seems to have closed the gap between them. The generic options (-O3 for g++ and -O2 for icc) will generally do most of the useful optimizations. But you should be aware that there are many additional optimization options that can improve particular codes, especially on a known architecture such as a Pentium 4. You should explore the man pages for the compiler to find out potential options.

In this session, we’ll take a look at a simple example of how coding the same operation different ways can make a dramatic difference in the execution time. For example, if we need to calculate the value of \(x^n\) where \(n\) is an integer, using the function \(\text{pow}(x,n)\) takes much longer than multiplying \(x\) together \(n\) times. That is because \(\text{pow}\) is a library function valid for any real value of \(n\) (i.e., any double). The operations needed in general (e.g., logarithms) take much more time than floating point multiplies and if the compiler doesn’t substitute for the general algorithm, there will be a big difference in times. We’ll see this in practice.

Since operations like \(\exp\), \(\sin\), and \(\cos\) are also expensive, if they are evaluated repeatedly with the same argument it is often efficient to use a “look-up” table. This is an array that is filled with the needed values once at the beginning of the program, and then just referenced later. An array lookup is much faster, as long as their is memory available for the array.
c. Profiling

The point of “profiling” is to identify areas of code that use the most overall time. There is no point in optimizing sections that use a small fraction of the total time, particularly if it causes the code to be less clear. For example, if a code spends roughly 90% of its time in one function and 10% in another, making the latter run ten times faster (which is an enormous improvement) will only make the code run 10% faster (e.g., 100 minutes before and 91 minutes after). Focus on the first function!

There is a standard GNU tool for profiling, called gprof. We’ll give it a try in this session, although you should be warned that it is not always helpful.

d. Using the Intel Compiler

So far we’ve used the GNU C++ compiler, called g++, exclusively. Another C++ compiler, supplied by Intel and optimized for Intel chips, is also available. There are several reasons it is useful to have an alternative compiler available:

- It may produce a faster executable through better optimization.
- There may be a bug in one of the compilers, which leads to incorrect results from your program.
- One compiler may give more useful warning messages than the other for debugging or verifying your code.

In this session, we’ll learn how to let up the Linux environment variables so we can use the Intel C++ compiler, called icc, in place of g++. 