14. 780.20 Session 14

a. Follow-ups from Sessions 12 and 13

- **Equilibration and Cooling.** Consider the 2D Ising model with ferromagnetic $J$. Here is an example of what you might find from running the `ising_model.cpp` code with $kT = 1$ three times:

![Graph showing energy vs. time for the Ising model](image)

We can see a rapid decrease in energy at small times $t$, which reaches a plateau by $t = 50$ about which successive configurations fluctuate. This is the equilibration or thermalization time (so these are the configurations we should throw out). But we see later that in one trial the energy drops to a lower plateau starting around $t = 100$ and another drops to the same lower plateau starting around $t = 500$. We might imagine the third trial will eventually drop down to the lower energy, but we can’t predict when. This problem of equilibrating to a higher energy (and there may be many other possibilities if we run for a long time) is not fixed by skipping a larger number of configurations at the beginning. (What would you pick? Up to $t = 100, 500, 2000$?) Rather, we give the system a chance to settle into the lowest energy by cooling. That is, we start the simulation at a higher temperature with a random configuration and equilibrate at that temperature. Then we use the final configuration as the starting configuration (rather than starting again with a random configuration) for a somewhat lower temperature until it equilibrates. And so on, until we get to the desired
temperature. How much to lower the temperature at each stage depends on the problem. (We’ll experiment with this rate in a slightly different context in Session 14 with simulated annealing.)

- **Variational Monte Carlo code.** The program `variational_SHO.cpp` in Session 13 uses the `VariationalMC` class to implement variational Monte Carlo for a toy one-dimensional harmonic oscillator example. The initial trial function included as a special case (when \( a = 1 \)) the actual ground-state wave function. Thus it was an unfair test that is almost guaranteed (once the Metropolis algorithm is optimized) to give a result very close to the exact value of 0.5. More realistic best answers with other trial wave functions that I tried included 0.59 at \( a \approx 0.65 \) with \( \psi(x) \propto e^{-(ax)^4} \) and 0.52 at \( a \approx 1.3 \) with \( \psi(x) \propto 1/\cosh(ax) \).

b. **Simulated Annealing**

Standard optimization methods are very good at finding local minima near where the minimization was started, but not usually good at finding the *global* minimum. Finding the global minimum of a function (such as the energy) is often (but not always) the goal. One strategy using conventional minimizers is to run multiple trials with the minimization started at different places in the parameter space (perhaps chosen at random) and then to keep the best minimum found of all the trials.

An alternative approach is to adapt the Metropolis Monte Carlo algorithm for generating a canonical Boltzmann distribution of configurations at a temperature \( T \) to mimic how physical systems find their ground states (i.e., the energy minimum at \( T = 0 \)). At high temperature (which means \( kT \) large compared to characteristic energy spacings), the equilibrium distribution will include many states. If the system is cooled *slowly*, then it will have the opportunity to explore many states and then settle into the lowest energy state as \( T \) goes to zero. This is called *annealing*. If the system is cooled quickly, it can get stuck in a state that is not the minimum (“quenching”); this is analogous to the routines we looked at in Session 10, which rapidly go “downhill” but only to local minima.

The strategy of *simulated annealing* is to mimic the annealing process by treating the function to be minimized as an energy (it might actually be an energy!), introducing an artificial temperature \( T \), and generating a sequence of states in a canonical distribution via the Metropolis algorithm. Then we lower the temperature according to a “schedule” (this just means according to a definite pattern) and let the system settle into (hopefully!) a configuration that minimizes the energy.

In practice this is not so easy:

- The problem needs to be cast into a form appropriate for this technique. This means we need to have a description of possible configurations of the system and then a way to change the configuration randomly (i.e., the analogs to specifying all the spins on a lattice and generating a new configuration by randomly flipping a spin). At the same time we need to identify an appropriate energy function to be minimized; this may be immediate, if the problem *is* to minimize a function, or less obvious, if the problem is to find a solution to a problem such as the Traveling Salesman [2].
• We need to devise an appropriate annealing schedule for the control parameter $T$. For example, do we change $T$ after 10 or 100 or 1000 or $? \text{ random changes in the configuration? And how much do we change it each time?}$ These are critical questions to the success of the procedure. If there is a physical connection to the problem, we may be able to use physics insight to determine the appropriate scales. Often it is more a trial-and-error procedure.

• For continuous control parameters, as opposed to combinatoric problems, the (common) possibility of long, narrow valleys in parameter space is a serious problem. If one takes steps at random, the most likely step will be uphill rather than along the valley. So one needs to modify the basic strategy, as discussed in Ref. [2].

We’ll apply simulated annealing to an artificial problem in this session, the global minimization of a one-dimensional function:

$$f(x) = e^{-(x-1)^2} \sin(8x),$$

(14.1)

which has multiple local minima. We’ll compare a standard minimization routine (from GSL) to simulated annealing. You’ll need to adjust the simulated annealing control parameters to make it work effectively. We won’t have time to consider a second, more realistic problem: the shape of molecules built from sodium (Na) and chlorine (Cl) atoms. This problem is described in a Session 10 handout from the book *An Introduction to Computational Physics* by T. Pang, in a section entitled “Geometric structures of multicharge clusters.” The idea is that one can write a potential energy function that depends on the relative positions of the elements of the clusters (here Na and Cl atoms). The parameters of the function are taken from experiment or theoretical calculations. The kinetic energy can be ignored, so the arrangement of the cluster is determined by minimizing the energy. There are many local minima corresponding to configurations that might be metastable but do not have the very lowest energy.

c. 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 (the compiler that is part of the gcc/g++ family). In the Linux C++ world, we have g++ and then commercial compilers, but sometimes the latter are freely available for academic use. In this class is the Intel C++ compiler icpc. It has a strong benchmarking record, particularly on certain platforms. The generic options (i.e., -O2 or -O3) 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.

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.

**Optimization Options for g++.** If you consult man g++ you’ll find a multitude of options tailoring the optimization of your code with the g++ compiler. The general options -00 through -03 turn on collections of these options:

-00 Do not optimize.

-01 These optimizations strive to reduce code size and execution time, using optimizations that do not take a lot of compilation time. It turns on these optimization flags:
  
  -fdefer-pop -fmerge-constants -fthread-jumps -ffunction-inlining -ftree-constant上线
  -fif-conversion -fif-conversion2 -fdelayed-branch
  -fguess-branch-probability -fcprop-registers

-02 Do all of the -01 optimizations plus many more:
  
  -fforce-mem -foptimize-sibling-calls -fstrength-reduce
  -fcse-follow-jumps -fcse-skip-blocks -frerun-cse-after-loop
  -frerun-loop-opt -fgcse -fgcse-lm -fgcse-sm -fgcse-las
  -fdelete-null-pointer-checks -fexpensive-optimizations -fregmove
  -fschedule-insns -fsched-insns2 -fsched-interblock
  -fsched-spec -fcaller-saves -fprefetch -freorder-blocks
  -freorder-functions -fstrict-aliasing -funit-at-a-time
  -fafinall-functions -fafinall-jumps -fafinall-loops -fafinall-labels
  -fcrossjumping

-03 Do all of the -02 optimizations as well as

  -finline-functions -fweb -frename-registers

The man pages describe each of these options, although the explanations are not very clear to the non-expert. The basic message is that a lot of processing is going on behind the scenes to try to make the code run faster. You should also consider the hardware-specific options, such as -march=i866 (for a pentium) or -march=opteron (for a 64-bit opteron).

d. **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 will try a quick demo of what it does.

### e. Using the Intel Compiler

So far we’ve used the GNU C++ compiler, called g++, exclusively. The C++ compiler icpc, supplied by Intel and optimized for Intel chips, is also available. (You can get it for free for a Linux computer from the Intel website.) 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 short, any important code should be compiled with more than one compiler.

### f. References

[1] M. Hjorth-Jensen, *Computational Physics*. These are notes from a course offered at the University of Oslo. See the 780.20 webpage for links.