

## 15. 780.20 Session 15

### a. Temperature Diffusion in One Dimension

The code `eqheat.cpp` simulates the time dependence of the temperature of a metal bar that is initially heated to  $100^\circ\text{C}$  and then allowed to cool with its ends kept at  $0^\circ\text{C}$  (the sides are assumed to be perfectly insulated so the heat flow is effectively one dimensional). The basic physics is that if there is a temperature gradient, then heat flows, but since energy is conserved there is a continuity equation. Let's derive the corresponding differential equation describing the temperature. In the following,  $\kappa = 0.12 \text{ cal}/(\text{s g cm }^\circ\text{C})$  is the thermal conductivity,  $c = 0.113 \text{ cal}/(\text{g }^\circ\text{C})$  is the specific heat, and  $\rho = 7.8 \text{ g}/\text{cm}^3$  is the mass density.

Consider a small piece of metal with constant cross section  $A$  and length  $\Delta x$ . The heat energy at time  $t$ ,  $\Delta Q(t)$ , is given by the specific heat times the mass of the piece times the temperature, or

$$\Delta Q(t) = [c\rho A \Delta x]T(x, t) + \mathcal{O}(\Delta x)^2. \quad (15.1)$$

(Dropping the  $(\Delta x)^2$  contribution will mean that we can evaluate the temperature at  $x$  or  $x + \Delta x$  or  $x + \Delta x/2$  and it doesn't matter.) Now we can write:

$$\text{Heat flow in at } x: \quad -\kappa \frac{\partial T(x, t)}{\partial x} \cdot A \quad (15.2)$$

$$\text{Heat flow out at } x + \Delta x: \quad +\kappa \frac{\partial T(x + \Delta x, t)}{\partial x} \cdot A. \quad (15.3)$$

The continuity equation equates the net heat flow to the time rate of change of the heat energy:

$$\frac{\partial \Delta Q}{\partial t} = c\rho A \Delta x \frac{\partial T(x, t)}{\partial t} = \kappa \left( \frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right) \cdot A. \quad (15.4)$$

Upon dividing by  $\Delta x$  (and other factors), we recognize the difference of first derivatives in  $x$  as a second derivative (up to  $(\Delta x)^2$  corrections). Thus, we obtain the diffusion equation

$$\frac{\partial T(x, t)}{\partial t} = \frac{\kappa}{c\rho} \frac{\left[ \frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right]}{\Delta x} = \frac{\kappa}{c\rho} \frac{\partial^2 T(x, t)}{\partial x^2} \quad (15.5)$$

in the limit that  $\Delta x$  goes to zero.

The code `eqheat.cpp` implements this equation by calculating the temperature change from time  $t$  to time  $t + \Delta t$  at each point  $x$  using

$$T(t + \Delta t, x) \approx T(t, x) + \Delta t \frac{\partial T(x, t)}{\partial t} + \mathcal{O}(\Delta t)^2 \quad (15.6)$$

and using the simplest finite-difference formulas to evaluate the second derivative in Eq. (15.5). To get started, we need to specify the temperature for  $0 \leq x \leq L$  for the initial time and also

the boundary conditions at  $x = 0$  and  $x = L$  for all times. This method might seem crude but foolproof, yet there is a major pitfall lurking: choosing values for  $\Delta t$  and  $\Delta x$ . Unless

$$\frac{\kappa}{c\rho} \frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{4}, \quad (15.7)$$

the numerical solution will not decay exponentially (see Landau and Paez, Chapter 26 for an explanation [1]). This means that decreasing  $\Delta t$  helps (up to a point, as usual), but if we decrease  $\Delta x$  to increase accuracy, we better decrease  $\Delta t$  quadratically. In practice, if there are not analytic solutions for guidance, one needs to try out different  $\Delta x$  and  $\Delta t$  values until the result is both stable *and* physically reasonable.

## 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 (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, which we'll explore in this session. It has a 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 icpc) 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 `pow(x,n)` takes much longer than multiplying  $x$  together  $n$  times. That is because `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 there 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 -O0 through -O3 turn on collections of these options:

-O0 Do not optimize.

-O1 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 -floop-optimize
-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 -fschedule-insns2 -fsched-interblock
-fsched-spec -fcaller-saves -fpeep-hole2 -freorder-blocks
-freorder-functions -fstrict-aliasing -funit-at-a-time
-falign-functions -falign-jumps -falign-loops -falign-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=i586` (for a pentium) or `-march=opteron` (for a 64-bit opteron).

### 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`. It is also accessible through Dev-C++.

### d. Using the Intel Compiler

So far we’ve used the GNU C++ compiler, called `g++`, exclusively. The C++ compiler `icc`, supplied by Intel and optimized for Intel chips, is also available. (You can get it for free for your own 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.

#### e. References

- [1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997).