

15. 780.20 Session 15

Session 15 is a brief look at partial differential equations or PDE's. We'll consider three very simple programs to calculate three linear PDE's with two variables. All are generalizable to more variables and more complicated boundary conditions. We give some brief background here while there is more detail in the excerpt from Landau/Paez [1].

a. Laplace's Equation in Two Dimensions

The code `laplace.cpp` solves for the electric potential $U(\mathbf{x})$ in a two-dimensional region with boundaries at fixed potentials (voltages). For a static potential in a region where the charge density $\rho_c(\mathbf{x})$ is identically zero, $U(\mathbf{x})$ satisfies Laplace's equation, $\nabla^2 U(\mathbf{x}) = 0$. In the x - y plane (i.e., assuming it is constant in the z direction), the equation reduces to

$$\frac{\partial^2 U(x, y)}{\partial x^2} + \frac{\partial^2 U(x, y)}{\partial y^2} = 0, \quad (15.1)$$

with boundary values enforced at the edges of the region. We'll solve this problem with a *relaxation method*. A PDE tells us locally how the value of the function is related to nearby values. Using finite difference approximations for the second derivatives we can derive the equation we need.

How do we derive a finite difference form for a second derivative? From a Taylor expansion, of course. Consider

$$U(x + \Delta x, y) = U(x, y) + \frac{\partial U}{\partial x} \Delta x + \frac{1}{2} \frac{\partial^2 U}{\partial x^2} (\Delta x)^2 + \mathcal{O}(\Delta x)^3 \quad (15.2)$$

$$U(x - \Delta x, y) = U(x, y) - \frac{\partial U}{\partial x} \Delta x + \frac{1}{2} \frac{\partial^2 U}{\partial x^2} (\Delta x)^2 - \mathcal{O}(\Delta x)^3, \quad (15.3)$$

which naturally sum to

$$U(x + \Delta x, y) + U(x - \Delta x, y) = 2U(x, y) + \frac{\partial^2 U}{\partial x^2} (\Delta x)^2 + \mathcal{O}(\Delta x)^4. \quad (15.4)$$

Doing the same thing in the y variable yields expressions for the derivatives in Eq. (15.1):

$$\frac{\partial^2 U}{\partial x^2} \approx \frac{U(x + \Delta x, y) + U(x - \Delta x, y) - 2U(x, y)}{(\Delta x)^2} \quad (15.5)$$

$$\frac{\partial^2 U}{\partial y^2} \approx \frac{U(x, y + \Delta y) + U(x, y - \Delta y) - 2U(x, y)}{(\Delta y)^2}. \quad (15.6)$$

Now take $\Delta x = \Delta y = \Delta$ and Eq. (15.1) gives us a relationship among neighboring points.

For our purposes we single out the point in the middle:

$$U(x, y) = \frac{1}{4}[U(x + \Delta x, y) + U(x - \Delta x, y) + U(x, y + \Delta y) + U(x, y - \Delta y)] + \mathcal{O}(\Delta^4). \quad (15.7)$$

The relaxation method consists of sweeping repeatedly through each point of the region (now divided into a grid) in turn, replacing its current value with a new one given by Eq. (15.7). We start with the fixed boundary values and some guess at the interior values. We keep sweeping until the values stop changing; at that point Laplace’s equation and the boundary conditions must be satisfied, so it *must* be the solution we seek! Instead of simply replacing the old value of $U(x, y)$ with $U_{\text{new}}(x, y)$ from Eq. (15.7), it may be more effective to introduce a “fraction” and take

$$U(x, y) = (1 - \text{fraction}) \times U_{\text{old}}(x, y) + \text{fraction} \times U_{\text{new}}(x, y) . \quad (15.8)$$

b. 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°C and then allowed to cool with its ends kept at 0°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 Δx . 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

$$\Delta Q(t) = [c\rho A \Delta x]T(x, t) + \mathcal{O}(\Delta x)^2 . \quad (15.9)$$

(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:

$$\text{Heat flow in at } x: \quad -\kappa \frac{\partial T(x, t)}{\partial x} \cdot A \quad (15.10)$$

$$\text{Heat flow out at } x + \Delta x: \quad +\kappa \frac{\partial T(x + \Delta x, t)}{\partial x} \cdot A . \quad (15.11)$$

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

$$\frac{\partial \Delta Q}{\partial t} = c\rho A \Delta x \frac{\partial T(x, t)}{\partial t} = \kappa \left(\frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right) \cdot A . \quad (15.12)$$

Upon dividing by Δx (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

$$\frac{\partial T(x, t)}{\partial t} = \frac{\kappa}{c\rho} \frac{\left[\frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right]}{\Delta x} = \frac{\kappa}{c\rho} \frac{\partial^2 T(x, t)}{\partial x^2} \quad (15.13)$$

in the limit that Δx goes to zero. [Note: if we put an i on the time side, we get the time-dependent Schrödinger equation.]

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

$$T(x, t + \Delta t) \approx T(x, t) + \Delta t \frac{\partial T(x, t)}{\partial t} + \mathcal{O}(\Delta t)^2 \quad (15.14)$$

and using the simplest finite-difference formulas, as in Eq. (15.5), to evaluate the second derivative in Eq. (15.13). The end result is

$$T(x, t + \Delta t) \approx T(x, t) + \frac{\kappa}{c\rho} \frac{\Delta t}{(\Delta x)^2} [T(x + \Delta x, t) + T(x - \Delta x, t) - 2T(x, t)]. \quad (15.15)$$

To get started, we need to specify the temperature for $0 \leq x \leq L$ for the initial time $t = 0$ and also the boundary conditions at $x = 0$ and $x = L$ for all times. Then we can step to $t = \Delta t$ for all x using Eq. (15.15), then $t = 2\Delta t$, and so on.

This method might seem crude but foolproof, yet there is a major pitfall lurking: choosing values for Δt and Δx . Unless

$$\frac{\kappa}{c\rho} \frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}, \quad (15.16)$$

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

c. Waves on a String

The code `eqstring.cpp` simulates the time dependence of a string of length l that is fixed at each end (defined as $x = 0$ and $x = l$) and plucked somehow at $t = 0$. The displacement $\psi(x, t)$ at each point x as a function of time t is described by a *wave equation*:

$$\frac{\partial^2 \psi(x, t)}{\partial x^2} = \frac{1}{c^2} \frac{\partial^2 \psi(x, t)}{\partial t^2}, \quad (15.17)$$

where c is the wave speed and the spatial boundary conditions are $\psi(0, t) = \psi(l, t)$. For a string of mass density (mass/length) ρ under tension τ , the wave speed is $c = \sqrt{\tau/\rho}$.

We proceed with a now familiar pattern: replace the derivatives in Eq. (15.17) by our favorite finite difference formula. We choose to step in time, so we solve for the term with $t + \Delta t$:

$$\psi(x, t + \Delta t) \approx 2\psi(x, t) - \psi(x, t - \Delta t) + \frac{c^2}{c'^2} [\psi(x + \Delta x, t) + \psi(x - \Delta x, t) - 2\psi(x, t)], \quad (15.18)$$

with $c' \equiv \Delta x/\Delta t$. Thus we can step forward in time for every x once we know the values of ψ at earlier times. To get started we need to know the initial $\psi(x, 0)$ (which is determined by how the string is plucked) and the initial value of $d\psi(x, 0)/dt$, which we take equal to 0 (the plucked string

is released from rest). The latter condition is implemented in the code by applying the central difference formula for the first derivative to derive a formula for the first time step. It is claimed that this method is stable *if*

$$c \leq c' = \frac{\Delta x}{\Delta t}. \quad (15.19)$$

You'll try this out "experimentally" in Session 15.

d. 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, 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.

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
```

-O2 Do all of the -O1 optimizations plus many more:

```

-fforce-mem -fopti- mize-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-ule-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

```

-O3 Do all of the -O2 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).

e. 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++.

f. 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.

g. References

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