

15. 780.20 Session 15

Session 15 has 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]. We also provide notes on the Schrödinger equation in momentum space as an example of solving integral equations with linear algebra and also calling Fortran library routines from C++.

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. Bound States in Momentum Space

The familiar time-independent Schrödinger equation in coordinate space for a *local* potential is an ordinary differential equation:

$$-\frac{\nabla^2}{2\mu}\psi_n(\mathbf{r}) + V(\mathbf{r})\psi_n(\mathbf{r}) = E_n\psi_n(\mathbf{r}), \quad (15.20)$$

where μ is the reduced mass (which is $M/2$ if we are considering two interacting particles of mass M each). For scattering states, where $E_n > 0$, any choice of E_n will give an acceptable solution (assuming $V(\mathbf{r}) \rightarrow 0$ sufficiently fast as $\mathbf{r} \rightarrow \infty$). For bound states, only discrete values of E_n yield normalizable wave functions, so we have an *eigenvalue* problem. In the more general (and less familiar case), the potential is *non-local* and we have an *integro-differential equation* to solve:

$$-\frac{\nabla^2}{2\mu}\psi_n(\mathbf{r}) + \int d^3\mathbf{r}' V(\mathbf{r}, \mathbf{r}')\psi_n(\mathbf{r}') = E_n\psi_n(\mathbf{r}). \quad (15.21)$$

If we recall the methods for solving the Schrödinger equation numerically that were discussed in Section 5d of the Session 5 notes, the direct solution as a differential equation would no longer be available but either the matrix diagonalization in coordinate representation or the expansion in an orthonormal basis could be extended to apply to Eq. (15.21). (In the former case, the potential would contribute everywhere in the matrix, since a non-local potential is not diagonal in coordinate representation. In the latter case, there is little difference with the local potential case we considered, since taking a matrix element of a non-local potential in a basis such as harmonic oscillators simply involves an extra integration.) Here we'll consider yet another option: momentum representation.

In momentum representation, the equation for the momentum space wave function $\psi_n(\mathbf{k})$ is (almost) *always* an integral equation (unless the potential is "separable"). Consider the abstract Schrödinger equation,

$$\hat{H}|\psi_n\rangle = \left(\frac{\hat{\mathbf{P}}^2}{2\mu} + \hat{V} \right) |\psi_n\rangle = E_n|\psi_n\rangle. \quad (15.22)$$

Now hit this on the left with $\langle \mathbf{k} |$ and insert

$$1 = \int d^3\mathbf{k}' |\mathbf{k}'\rangle \langle \mathbf{k}'| \quad (15.23)$$

to obtain

$$\frac{k^2}{2\mu} \langle \mathbf{k} | \psi_n \rangle + \int d^3\mathbf{k}' \langle \mathbf{k} | V | \mathbf{k}' \rangle \langle \mathbf{k}' | \psi_n \rangle = E_n \langle \mathbf{k} | \psi_n \rangle \quad (15.24)$$

or, in an alternative notation for the same thing,

$$\frac{k^2}{2\mu}\psi_n(\mathbf{k}) + \int d^3\mathbf{k}' V(\mathbf{k}, \mathbf{k}')\psi_n(\mathbf{k}') = E_n\psi_n(\mathbf{k}) . \quad (15.25)$$

If we expand in a partial wave basis (check your favorite quantum book!), then the resulting one-dimensional equation in the l^{th} partial wave takes the form

$$\frac{k^2}{2\mu}\psi_n(k) + \frac{2}{\pi} \int_0^\infty V(k, k')\psi_n(k') k'^2 dk' = E_n\psi_n(k) , \quad (15.26)$$

where $k \equiv |\mathbf{k}|$ and we omit l labels on the potential and wave functions.

The potential in partial waves is the “Bessel transform” of the full potential (why not the Fourier transform?):

$$V(k, k') = \int_0^\infty r dr \int_0^\infty r' dr' j_l(kr')V(r', r)j_l(k'r) , \quad (15.27)$$

which reduces for a local potential to

$$V(k, k') = \int_0^\infty r^2 dr j_l(kr)V(r)j_l(k'r) . \quad (15.28)$$

Recall that the first two spherical Bessel functions are

$$j_0(z) = \frac{\sin z}{z} , \quad j_1(z) = \frac{\sin z}{z^2} - \frac{\cos z}{z} , \quad (15.29)$$

so for $l = 0$, the potential is simply

$$V(k, k')_{l=0} = \frac{1}{kk'} \int_0^\infty dr \sin(kr)V(r) \sin(k'r) . \quad (15.30)$$

e. Numerical Solution

So how do we solve for the E_n 's and corresponding $\psi_n(k)$'s in Eq. (15.26)? As we've done before, we discretize it (that is, break up the continuous range in k into mesh points) and turn it into a matrix eigenvalue problem. Thus, if we have an integration rule (such as Gaussian quadrature) that performs an integral from 0 to ∞ as a sum over N points $\{k_i\}$ with weights $\{w_i\}$, then the integral over the potential becomes

$$\int_0^\infty k'^2 dk' V(k, k')\psi_n(k') \approx \sum_{j=0}^{N-1} w_j k_j^2 V(k, k_j)\psi_n(k_j) . \quad (15.31)$$

Thus the Schrödinger equation becomes

$$\frac{k_i^2}{2\mu}\psi_n(k_i) + \frac{2}{\pi} \sum_{j=0}^{N-1} w_j k_j^2 V(k_i, k_j)\psi_n(k_j) = E_n\psi_n(k_i) , \quad i = 0, \dots, N-1 . \quad (15.32)$$

This is just the matrix problem

$$\sum_j H_{ij} [\psi_n]_j = E_n [\psi_n]_i, \quad (15.33)$$

with

$$H_{ij} \equiv \frac{k_i^2}{2\mu} \delta_{ij} + \frac{2}{\pi} V(k_i, k_j) k_j^2 w_j, \quad i, j = 0, \dots, N-1. \quad (15.34)$$

We can turn this over to a packaged matrix eigenvalue routine and get the eigenvalues and eigenvectors directly.

Note, however, that the matrix is *not* symmetric. This is not a problem in principle, since there are routines that can solve a general non-symmetric eigenvalue problem (e.g., in the LAPACK subroutine library or the latest release of GSL). However, a better idea is to turn the problem into a symmetric matrix problem. We do this by multiplying Eq. (15.33) by $k_i \sqrt{w_i}$ to get:

$$\sum_j \tilde{H}_{ij} [\tilde{\psi}_n]_j = E_n [\tilde{\psi}_n]_i, \quad (15.35)$$

where

$$[\tilde{\psi}_n]_i \equiv k_i \sqrt{w_i} [\psi_n]_i. \quad (15.36)$$

This means that \tilde{H}_{ij} is

$$\tilde{H}_{ij} \equiv \frac{k_i^2}{2\mu} \delta_{ij} + \frac{2}{\pi} k_i \sqrt{w_i} V(k_i, k_j) k_j \sqrt{w_j}, \quad i, j = 0, \dots, N-1, \quad (15.37)$$

so we now have a symmetric problem with the same eigenvalues. Note also that if the vector $[\tilde{\psi}_n]_i$ is normalized so that (assuming it is real)

$$[\tilde{\psi}_n] \cdot [\tilde{\psi}_n] = \sum_i [\tilde{\psi}_n]_i^2 = 1, \quad (15.38)$$

then

$$1 = \sum_i k_i^2 w_i [\psi_n]_i^2 \longrightarrow \int_0^\infty k^2 dk |\psi(k)|^2, \quad (15.39)$$

so the continuum version is normalized as well.

f. Delta-Shell Potential

The potential we'll use in this session is the “delta-shell” potential, which in the coordinate representation is

$$V(r) = \frac{\lambda}{2\mu} \delta(r-b), \quad (15.40)$$

where μ is the reduced mass of the particles interacting via V (or just think of μ as the mass of a particle in the external potential V). Note that this is *not* a delta function at the origin; the potential is zero unless the particles are separated precisely by a distance $r = b$. So if we have a force that effectively acts over a very short but nonzero range of distances, this might be a reasonable

(although crude) representation. Besides the mass, the parameters are the range b and the strength λ . From Eq. (15.40) you should be able to directly determine the units of λ .

The s-wave ($l = 0$) Schrödinger equation has (at most) one bound-state (that is, $E < 0$) solution. If we define κ by writing the bound-state energy as

$$E = -\frac{\kappa^2}{2\mu}, \quad (15.41)$$

the value of κ is determined by the solution to the transcendental equation

$$e^{-2\kappa b} - 1 = \frac{2\kappa}{\lambda} \quad (l = 0). \quad (15.42)$$

For general l , the bound-state κ is the solution to [1]

$$1 - \frac{\lambda}{i\kappa} (i\kappa b)^2 j_l(i\kappa b) [n_l(i\kappa b) - i j_l(i\kappa b)]. \quad (15.43)$$

Can you derive either of these results? Is there always one bound state?

The delta-shell potential is trivial to convert to momentum space:

$$V(k', k) = \int_0^\infty r^2 dr j_l(k'r) \frac{\lambda}{2\mu} \delta(r - b) j_l(\kappa r) = \frac{\lambda b^2}{2\mu} j_l(k'b) j_l(kb), \quad (15.44)$$

where l is the angular momentum state we are considering. Note that this is not a very well-behaved function in momentum space! That means you may have to be clever in doing a numerical integral. The wave function of the $l = 0$ bound state in coordinate space is

$$\psi_0(r) = \int_0^\infty k^2 dk \psi_0(k) j_0(kr) \propto \begin{cases} e^{-\kappa r} - e^{\kappa r}, & \text{for } r < b, \\ e^{-\kappa r}, & \text{for } r > b. \end{cases} \quad (15.45)$$

g. Calling Fortran from C++

We'll use the example of calling a LAPACK fortran library machine from C++. In order to make the call as similar as possible to the fortran, we'll use C++ "references" rather than pointers. A reference is an alias to a variable (something like a short-cut in Windows). We declare a pointer using a `*`; a reference is declared using a `&`. Here's what we need to do:

1. Add an underscore to the lowercase name of the Fortran routine, e.g., DGEEV becomes `dgeev_`.
2. Include a prototype declaration of the Fortran subrouine. Put `extern "C" { }` around the prototype. For example (note that most are declared `const`):

```
extern "C" {
    void dgeev_(const char &JOBVL, const char &JOBVR,
               const int &dimension1, double Hmat_passed[],
               const int &dimension2, double Eigval_real[], double Eigval_imag[],
               double Eigvec_left[][1], const int &LDVL,
```

```

    double Eigvec_right_passed[], const int &LDVR,
    double WORK[], const int &LWORK, int &INFO);
}

```

3. The variable type used in C++ should match the Fortran type, as illustrated in this chart (which uses the variable definitions in the Session 16 code `deltashell_boundstates.cpp`):

Fortran		C++		
defined	passed	defined	prototype	passed
CHARACTER*1 JOBVL	JOBVL	char JOBVL	char &JOBVL	JOBVL
INTEGER N	N	int dimension	int &dimension	dimension
REAL*8 WR(*)	WR	double* Eigval_real = new double [dimension]	double Eigval_real[]	Eigval_real

4. Fortran arrays start at 1 rather than 0. This is not a problem when passing arrays from C++ to Fortran. Simply fill the C++ array as usual (starting from 0), pass the pointer to Fortran, and it will be interpreted as starting from 1.
5. The Fortran array element `A(3,5)` is `A[4][2]` in C++ (subtract one for zero base indexing and reverse the order of subscripts). You must fill your C++ arrays accordingly.
6. If you use `g++`, compile the C++ parts as usual and then link using
`-lm -lblas -llapack -lg2c`
 (For libraries other than LAPACK, link with the appropriate library names; you'll always need `-lg2c`, however.)

h. Dynamically Allocating Space for Arrays

Suppose we want to allocate `f[i]` with space for `maxsize` elements. Then:

```
double* f = new double [maxsize]
```

after which we can refer to `f[0]`, `f[1]`, ... `f[maxsize-1]`. To deallocate `f` and free the memory:

```
delete [] f
```

(note that no number appears between the `[]`'s).

i. References

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