16. 780.20 Session 16

a. Follow-ups to Session 15

- **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, \( \kappa = 0.12 \text{cal/(s g cm}^\circ \text{C)} \) is the thermal conductivity, \( c = 0.113 \text{cal/(g}^\circ \text{C)} \) is the specific heat, and \( \rho = 7.8 \text{g/cm}^3 \) 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

\[
\Delta Q(t) = [c \rho A \Delta x] T(x, t) + O(\Delta x)^2 .
\] (16.1)

(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:

Heat flow in at \( x \): \[ -\kappa \frac{\partial T(x, t)}{\partial x} \cdot A \] (16.2)

Heat flow out at \( x + \Delta x \): \[ +\kappa \frac{\partial T(x + \Delta x, t)}{\partial x} \cdot A . \] (16.3)

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

\[
\frac{\partial \Delta Q}{\partial t} = c \rho A \Delta x \frac{\partial T(x, t)}{\partial t} = \kappa \left( \frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right) \cdot A .
\] (16.4)

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

\[
\frac{\partial T(x, t)}{\partial t} = \frac{\kappa}{c \rho A} \left( \frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right) \cdot A .
\] (16.5)

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

\[
T(t + \Delta t, x) \approx T(t, x) + \Delta t \frac{\partial T(x, t)}{\partial t} + O(\Delta t)^2
\] (16.6)

and using the simplest finite-difference formulas to evaluate the second derivative in Eq. (16.5).

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

\[
\frac{\kappa}{c\rho} \frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{4},
\]

(16.7)

the numerical solution will not decay exponentially (see Landau and Paez, Chapter 26 for an explanation [2]). 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.

- **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` - `floop-optimize`
  - `fif-conversion` - `fif-conversion2` - `fdelayed-branch`
  - `fguess-branch-probability` - `fprop-registers`

- **-02** Do all of the -01 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`

- **-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).
b. Bound States in Momentum Space

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

\[-\frac{\nabla^2}{2\mu}\psi_n(r) + V(r)\psi_n(r) = E_n\psi_n(r) , \tag{16.8}\]

where \(\mu\) 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 \(V(r) \to 0\) sufficiently fast as \(r \to \infty\)). 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:

\[-\frac{\nabla^2}{2\mu}\psi_n(r) + \int d^3r' V(r, r')\psi_n(r') = E_n\psi_n(r) . \tag{16.9}\]

In momentum space, the equation for the momentum space wave function \(\psi_n(k)\) is (almost) always an integral equation (unless the potential is “separable”). Consider the abstract Schrödinger equation,

\[\hat{H}\langle \psi_n \rangle = \left(\frac{\hat{P}^2}{2\mu} + \hat{V}\right)\langle \psi_n \rangle = E_n\langle \psi_n \rangle . \tag{16.10}\]

Now hit this on the left with \(\langle k |\) and insert

\[1 = \int d^3k' |k'|\langle k'| \]

(16.11)

to obtain

\[\frac{k^2}{2\mu}\langle k |\psi_n \rangle + \int d^3k' \langle k |V| k' \rangle\langle k'| \psi_n \rangle = E_n\langle k |\psi_n \rangle \tag{16.12}\]

or, in an alternative notation for the same thing,

\[\frac{k^2}{2\mu}\psi_n(k) + \int d^3k' V(k, k')\psi_n(k') = E_n\psi_n(k) . \tag{16.13}\]

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

\[\frac{k^2}{2\mu}\psi_n(k) + \frac{2}{\pi} \int_0^\infty V(k, k')\psi_n(k') k'^2 dk' = E_n\psi_n(k) , \tag{16.14}\]

where \(k \equiv |k|\) and we omit \(l\) labels.

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

\[V(k, k') = \int_0^\infty r dr \int_0^\infty r' dr' j_l(kr')V(r', r)j_l(k'r) , \tag{16.15}\]
which reduces for a local potential to
\[ V(k, k') = \int_0^\infty r^2 dr \, j_i(kr)V(r)j_i(k'r) \, . \] (16.16)

Recall that the first two spherical Bessel functions are
\[ j_0(z) = \frac{\sin z}{z} \, , \quad j_1(z) = \frac{\sin z}{z^2} - \frac{\cos z}{z} \, , \] (16.17)
so for \( l = 0 \), the potential is simply
\[ V(k, k')_{l=0} = \frac{1}{kk'} \int_0^\infty dr \, \sin(kr)V(r)\sin(k'r) \, . \] (16.18)

c. Numerical Solution

So how do we solve for the \( E_n \)'s and corresponding \( \psi_n(k) \)'s in Eq. (16.14)? 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 \( \infty \) as a sum over \( N \) points \( \{k_i\} \) with weights \( \{w_i\} \), then the integral over the potential becomes
\[ \int_0^\infty k'k^2 dk' \psi_n(k') \approx \sum_{j=0}^{N-1} w_j k_j^2 V(k_i, k_j) \psi_n(k_j) \, . \] (16.19)

Thus the Schrödinger equation becomes
\[ \frac{k_i^2}{2\mu} \psi_n(k_i) + \frac{2}{\pi} \sum_{j=0}^{N-1} w_j k_j^2 V(k_i, k_j) \psi_n(k_j) = E_n \psi_n(k_i) \, , \quad i = 0, \ldots, N - 1 \, . \] (16.20)

This is just the matrix problem
\[ \sum_j H_{ij} \psi_n[j] = E_n \psi_n[j] \, , \] (16.21)
with
\[ H_{ij} \equiv \frac{k_i^2}{2\mu} \delta_{ij} + \frac{2}{\pi} \frac{V(k_i, k_j)}{k_j^2} w_j \, , \quad i, j = 0, \ldots, N - 1 \, . \] (16.22)

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, so we can’t use the simple GSL routines. Instead we’ll use a general eigenvalue solver from the LAPACK subroutine library. There are versions of LAPACK for C and C++, but the most robust version is written in Fortran. So we’ll use this problem as an excuse to see how to call Fortran routines from C++. 
d. Delta-Shell Potential

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

\[ V(r) = \frac{\lambda}{2\mu} \delta(r - b) , \]  

(16.23)

where \( \mu \) is the reduced mass of the particles interacting via \( V \) (or just think of \( \mu \) 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 \( \lambda \). From Eq. (16.23) you should be able to directly determine the units of \( \lambda \).

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

\[ E = -\frac{\kappa^2}{2\mu} , \]  

(16.24)

the value of \( \kappa \) is determined by the solution to the transcendental equation

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

(16.25)

For general \( l \), the bound-state \( \kappa \) is the solution to \( [1] \)

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

(16.26)

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

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

\[ V(k', k) = \int_0^\infty r^2 dr \frac{\lambda}{2\mu} \delta(r - b) j_l(kr) = \frac{\lambda b^2}{2\mu} j_l(kb) j_l(kb) , \]  

(16.27)

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

\[ \psi_0(r) = \int_0^\infty k^2 dk \psi_0(k) j_0(kr) \propto \left\{ \begin{array}{ll} e^{-\kappa r} - e^{\kappa r} , & \text{for } r < b , \\ e^{-\kappa r} , & \text{for } r > b . \end{array} \right. \]  

(16.28)

e. 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 subroutine. Put extern "C" { } around the prototype. For example (note that most are declared const):

```c
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[][], 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`):

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>defined</td>
<td>passed</td>
</tr>
<tr>
<td>CHARACTER*1 JOBVL</td>
<td>JOBVL char</td>
</tr>
<tr>
<td>INTEGER N</td>
<td>N int</td>
</tr>
<tr>
<td>REAL<em>8 WR(</em>)</td>
<td>WR double* Eigval_real</td>
</tr>
<tr>
<td></td>
<td>= new double [dimension]</td>
</tr>
<tr>
<td></td>
<td>double Eigval_real[]</td>
</tr>
<tr>
<td></td>
<td>Eigval_real</td>
</tr>
</tbody>
</table>

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.)

f. Dynamically Allocating Space for Arrays

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

```c
  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:

```c
  delete [] f
```

(note that no number appears between the []'s). See the `deltashell_boundstates.cpp` code for an example of how to allocate and deallocate two-dimensional arrays.

g. References