Brief Notes on Solving PDE's and Integral Equations

In these notes adapted from the Physics 780 Computational Physics course, we'll consider three simple programs to calculate three linear partial differential equations (PDE's) with two independent variables using finite difference approximations. [Note: finite element methods can be more powerful.] Code listings are included in these notes. All are generalizable to more variables and more complicated boundary conditions. We give some brief background here while there is more detail in Refs. [1], [2] and [3]. We also provide notes on the Schrödinger equation in momentum space as an example of solving integral equations with gaussian quadrature and linear algebra.

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

$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = 0 \; , \tag{1}$$

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

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

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

which naturally sum to

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

Doing the same thing in the y variable yields expressions for the derivatives in Eq. (1):

$$\frac{\partial^2 U}{\partial x^2} \approx \frac{U(x+\Delta x, y) + U(x-\Delta x, y) - 2U(x,y)}{(\Delta x)^2} \tag{5}$$

$$\frac{\partial^2 U}{\partial y^2} \approx \frac{U(x, y+\Delta y) + U(x, y-\Delta y) - 2U(x,y)}{(\Delta y)^2} \; . \tag{6}$$

Now take $\Delta x = \Delta y = \Delta$ and Eq. (1) gives us a relationship among neighboring points.

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

$$U(x,y) = \frac{1}{4}[U(x+\Delta x, y) + U(x-\Delta x, y) + U(x, y+\Delta y) + U(x, y-\Delta y)] + \mathcal{O}(\Delta^4) \; . \tag{7}$$

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. (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 $U_{\rm new}(x, y)$ from Eq. (7), it is usually more effective to introduce a "fraction" and take

$$U(x, y) = (1 - \text{fraction}) \times U_{\rm old}(x, y) + \text{fraction} \times U_{\rm new}(x, y) \; . \tag{8}$$

Besides the `laplace.cpp` code, there are several MATLAB versions available (`laplace_relax.m`, `laplace_relax_pbc.m`, and `laplace_relax_random.m`) that use different boundary conditions and different initial conditions.

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

$$\Delta Q(t) = [c\,\rho A\,\Delta x] T(x, t) + \mathcal{O}(\Delta x)^2 \; . \tag{9}$$

(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:

$$\text{Heat flow in at } x: \qquad -\kappa \, \frac{\partial T(x, t)}{\partial x} \cdot A \tag{10}$$

$$\text{Heat flow out at } x + \Delta x: \qquad +\kappa \, \frac{\partial T(x + \Delta x, t)}{\partial x} \cdot A \; . \tag{11}$$

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

$$\frac{\partial \Delta Q}{\partial t} = c\,\rho A\,\Delta x \frac{\partial T(x, t)}{\partial t} = \kappa \left(\frac{\partial T(x + \Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right) \cdot A \; . \tag{12}$$

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

$$\frac{\partial T(x, t)}{\partial t} = \frac{\kappa}{c\rho} \frac{\left[\frac{\partial T(x+\Delta x, t)}{\partial x} - \frac{\partial T(x, t)}{\partial x} \right]}{\Delta x} = \frac{\kappa}{c\rho} \frac{\partial^2 T(x, t)}{\partial x^2} \tag{13}$$

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

$$T(x, t + \Delta t) \approx T(x, t) + \Delta t \frac{\partial T(x, t)}{\partial t} + \mathcal{O}(\Delta t)^2 \tag{14}$$

and using the simplest finite-difference formulas, as in Eq. (5), to evaluate the second derivative in Eq. (13). The end result is

$$T(x, t + \Delta t) \approx T(x, t) + \frac{\kappa}{c\rho} \frac{\Delta t}{(\Delta x)^2} [T(x + \Delta x, t) + T(x - \Delta x, t) - 2T(x, t)] \, . \tag{15}$$

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), 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

$$\frac{\kappa}{c\rho} \frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2} \, , \tag{16}$$

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*:

$$\frac{\partial^2 \psi(x, t)}{\partial x^2} = \frac{1}{c^2} \frac{\partial^2 \psi(x, t)}{\partial t^2} \, , \tag{17}$$

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. (17) by our favorite finite difference formula. We choose to step in time, so we solve for the term with $t + \Delta t$:

$$\psi(x, t + \Delta t) \approx 2\psi(x, t) - \psi(x, t - \Delta t) + \frac{c^2}{c'^2} [\psi(x + \Delta x, t) + \psi(x - \Delta x, t) - 2\psi(x, t)] \, , \tag{18}$$

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*

$$c \leq c' = \frac{\Delta x}{\Delta t} \; . \tag{19}$$

For more details on stability conditions, see Ref. [3].

d. PDE C++ Code Listing: laplace.cpp

```
//*****************************************************************************
//  laplace.cpp:   Solution of Laplace equation with finite differences
//
//  based on: laplace.c in "Projects in Computational Physics" by Landau/Paez,
//              ch. 25, code copyrighted by RH Landau
//  programmer: Dick Furnstahl furnstahl.1@osu.edu 12/04/02 (C), 01/04/03 (C++)
//                                                 05/15/08 (minor upgrades)
//  notes:  uses gnuplot 3D grid format. plot with:
//   gnuplot> splot "laplace.dat" with lines
//*****************************************************************************
#include <iostream>
#include <fstream>

int
main ()
{
  const int size = 40; // grid size (grid points)
  const int num_iter = 500;             // number of interations

  double potl[size][size];      // the electric potential

  std::ofstream outfile ("laplace.dat"); // open an output file stream

  for (int i = 0; i < size; i++) // set up boundary conditions
  {
    potl[i][0] = 100.0; // "top" edge is at 100 V
    for (int j = 1; j < size; j++)
    {
      potl[i][j] = 0;         // rest of grid is at 0 V
    }
  }

  for (int iter = 0; iter < num_iter; iter++) // iterations
  {
    for (int i = 1; i < (size - 1); i++) // x-direction
    {
      for (int j = 1; j < (size - 1); j++) // y-direction
```

```
      {
        potl[i][j] = 0.25 * ( potl[i+1][j] + potl[i-1][j]
                              +  potl[i][j+1] + potl[i][j-1] );
       }
    }
  }

  for (int i = 0; i < size; i++) // write data in gnuplot 3D format
  {
    for (int j = 0; j < size; j++)
    {
      outfile << potl[i][j] << std::endl; // gnuplot 3D grid format
    }
    outfile << std::endl; // empty line for gnuplot
  }
  std::cout << std::endl << "data stored in laplace.dat" << std::endl;
  outfile.close ();
  return (0);
}
```

e. PDE C++ Code Listing: eqheat.cpp

```
//*****************************************************************************
//  eqheat.cpp:   Solution of heat equation using with finite differences
//
//  based on: eqheat.c in "Projects in Computational Physics" by Landau/Paez,
//              ch. 26, code copyrighted by RH Landau
//  programmer: Dick Furnstahl furnstahl.1@osu.edu 12/04/02 (C), 01/04/03 (C++),
//                                                 03/05/06 (minor upgrades)
//  notes:  uses gnuplot 3D grid format. plot with (might want "with lines"):
//   gnuplot> set view 60,150; set xrange [0:100] reverse; splot "eqheat.dat"
//*****************************************************************************
#include <iostream>
#include <fstream>

int
main ()
{
  const double length = 150.;    // length in cm
  const int size = 151; // grid size
  const int num_steps = 30000; // number of timesteps
  const double conductivity = 0.12;   // units: (cal/s cm degree C)
  const double specific_heat = 0.113; // units: (cal/ g degree C)
  const double rho = 7.8; // density in (g/cm^3)

  double T[size][2]; // temperature at x and two times
  std::ofstream outfile ("eqheat.dat"); // open an output file stream
```

```
  double delta_x = length/double(size-1);   // mesh spacing in x

  for (int i = 0; i < size; i++)
  {
    T[i][0] = 100.; // at t=0, all points are at 100 C
  }
  for (int j = 0; j < 2; j++) // except the endpoints, which are 0 C
  {
    T[0][j] = T[size-1][j] = 0.;
  }

  double constant = conductivity / (specific_heat*rho) / (delta_x*delta_x);
  for (int i = 1; i <= num_steps; i++) // loop over num_steps timesteps
  {
    for (int j = 1; j < (size - 1); j++) // loop over space
    {
      T[j][1] = T[j][0]
                + constant * (T[j+1][0] + T[j-1][0] - 2.0 * T[j][0]);
    }
    if ((i % 1000 == 0) || (i == 1)) // save every 1000 time steps
    {
      for (int j = 0; j < size; j++)
      {
        outfile << T[j][1] << std::endl; // gnuplot 3D grid format
      }
      outfile << std::endl; // empty line for gnuplot
    }
    for (int j = 0; j < size; j++)
    {
      T[j][0] = T[j][1]; // shift new values to old
    }
  }
  std::cout << std::endl << "data stored in eqheat.dat" << std::endl;
  outfile.close ();
  return (0);
}
```

f. PDE C++ Code Listing: eqstring.cpp

```
//*****************************************************************************
//  eqstring.cpp:   Solution of wave equation using time stepping
//
//  based on: eqstring.c in "Projects in Computational Physics"
//              by Landau/Paez,  ch. 26, code copyrighted by RH Landau
//
//  programmer: Dick Furnstahl furnstahl.1@osu.edu 12/04/02 (C), 01/04/03 (C++)
//                                                 05/15/08 (major upgrades)
//  notes:  uses gnuplot 3D grid format. plot with
```

```
//     gnuplot> set view 60,150; set xrange [0:100] reverse
//     gnuplot> splot "eqstring.dat" with lines
//*****************************************************************************
#include <iostream>
#include <fstream>

const double length = 1;        // length of string in meters
const double rho = 0.01; // density per length in kg/m
const double tension = 40.0; // tension in Newtons
const int num_pts = 101;        // # of x points
const int time_steps = 100; // no. of time steps

int
main ()
{
  double psi[num_pts][3];  // psi[i][j] is displacement at point i for
                           //  time j=0 (past), time j=1 (present), and
                           //  time j=2 (future)

  double c_speed_sq = tension/rho;   // speed^2 of the wave ("c")
  double c_prime_sq = c_speed_sq;    // c' = Delta x / Delta t

  std::ofstream outfile ("eqstring.dat"); // open an output file stream

  // set initial configuration for plucked string by specifying the
  //  displacement at each x point
  int i_pluck = 81;
  for (int i = 0; i < i_pluck; i++)
  {
    // from book:  psi(x,t=0) = 1.25 x/length for x < 0.8 length
    psi[i][0] = 1.25 * double(i) / double(num_pts-1);
  }
  for (int i = i_pluck; i < num_pts; i++)
  {
    // from book:  psi(x,t=0) = 5(1-x/l) for x > 0.8 length
    psi[i][0] = 5. * (1. - double(i) / double(num_pts-1) );
  }

  // do the first time step of all x except the ends
  for (int i = 1; i < num_pts-1; i++)
  {
    psi[i][1] = psi[i][0] + (c_speed_sq/c_prime_sq)/2.0
                * ( psi[i+1][0] + psi[i-1][0] - 2.0*psi[i][0] );
  }

  psi[0][1] = psi[100][1] = 0.; // fixed boundary conditions
  psi[0][2] = psi[100][2] = 0.; // fixed boundary conditions

  for (int k = 1; k < time_steps; k++) // all later time steps
```

```
  {
    for (int i = 1; i < num_pts-1; i++)
    {
      psi[i][2] = 2.0*psi[i][1] - psi[i][0] + (c_speed_sq/c_prime_sq)
                  * ( psi[i+1][1] + psi[i-1][1] - 2.0*psi[i][1] );
    }
    for (int i = 0; i < num_pts; i++) // shift new values to old
    {
      psi[i][0] = psi[i][1];
      psi[i][1] = psi[i][2];
    }
    if ((k % 5) == 0) // print every 5th point
    {
      for (int i = 0; i < num_pts; i++)
      {
        outfile << psi[i][2] << std::endl;    // gnuplot 3D grid format
      }
      outfile << std::endl; // empty line for gnuplot
    }
  }
  std::cout << std::endl << "data stored in eqstring.dat" << std::endl;
  outfile.close ();
  return (0);
}
```

g. Bound States in Momentum Space

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

$$-\frac{\nabla^2}{2\mu}\psi_n(\mathbf{r}) + V(\mathbf{r})\psi_n(\mathbf{r}) = E_n\psi_n(\mathbf{r}) \ , \tag{20}$$

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 $V(\mathbf{r}) \to 0$ sufficiently fast as $\mathbf{r} \to \infty$). 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:

$$-\frac{\nabla^2}{2\mu}\psi_n(\mathbf{r}) + \int d^3\mathbf{r}'\, V(\mathbf{r},\mathbf{r}')\psi_n(\mathbf{r}') = E_n\psi_n(\mathbf{r}) \ . \tag{21}$$

If we think about possible methods for solving the Schrödinger equation numerically, a direct solution as a differential equation is no longer available but we could apply a matrix diagonalization in coordinate representation or use an expansion in an orthonormal basis with Eq. (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, 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,

$$\widehat{H}|\psi_n\rangle = \left(\frac{\widehat{\mathbf{P}}^2}{2\mu} + \widehat{V}\right)|\psi_n\rangle = E_n|\psi_n\rangle \ . \tag{22}$$

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

$$1 = \int d^3\mathbf{k}' \, |\mathbf{k}'\rangle\langle\mathbf{k}'| \tag{23}$$

to obtain

$$\frac{k^2}{2\mu}\langle\mathbf{k}|\psi_n\rangle + \int d^3\mathbf{k}' \, \langle\mathbf{k}|V|\mathbf{k}'\rangle\langle\mathbf{k}'|\psi_n\rangle = E_n\langle\mathbf{k}|\psi_n\rangle \tag{24}$$

or, in an alternative notation for the same thing,

$$\frac{k^2}{2\mu}\psi_n(\mathbf{k}) + \int d^3\mathbf{k}' \, V(\mathbf{k},\mathbf{k}')\psi_n(\mathbf{k}') = E_n\psi_n(\mathbf{k}) \ . \tag{25}$$

If we expand in a partial wave basis (this means to use the spherical coordinate basis with spherical harmonics), then the resulting one-dimensional equation in the l^{th} partial wave takes the form

$$\frac{k^2}{2\mu}\psi_n(k) + \frac{2}{\pi}\int_0^\infty V(k,k')\psi_n(k') \, k'^2 dk' = E_n\psi_n(k) \ , \tag{26}$$

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?):

$$V(k,k') = \int_0^\infty r \, dr \int_0^\infty r' \, dr' \, j_l(kr')V(r',r)j_l(k'r) \ , \tag{27}$$

which reduces for a local potential to

$$V(k,k') = \int_0^\infty r^2 dr \, j_l(kr)V(r)j_l(k'r) \ . \tag{28}$$

Recall that the first two spherical Bessel functions are

$$j_0(z) = \frac{\sin z}{z} \ , \qquad j_1(z) = \frac{\sin z}{z^2} - \frac{\cos z}{z} \ , \tag{29}$$

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

$$V(k,k')_{l=0} = \frac{1}{kk'}\int_0^\infty dr \, \sin(kr)V(r)\sin(k'r) \ . \tag{30}$$

h. Numerical Solution

So how do we solve for the E_n's and corresponding $\psi_n(k)$'s in Eq. (26)? An effective strategy is to 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

$$\int_0^\infty k'^2 dk'\, V(k,k')\psi_n(k') \approx \sum_{j=0}^{N-1} w_j k_j^2\, V(k,k_j)\psi_n(k_j)\ . \tag{31}$$

Thus the Schrödinger equation becomes

$$\frac{k_i^2}{2\mu}\psi_n(k_i) + \frac{2}{\pi}\sum_{j=0}^{N-1} w_j k_j^2\, V(k_i,k_j)\psi_n(k_j) = E_n\psi_n(k_i)\ , \quad i=0,\cdots,N-1\ . \tag{32}$$

This is just the matrix problem

$$\sum_j H_{ij}\,[\psi_n]_j = E_n\,[\psi_n]_i\ , \tag{33}$$

with

$$H_{ij} \equiv \frac{k_i^2}{2\mu}\delta_{ij} + \frac{2}{\pi}V(k_i,k_j)\,k_j^2 w_j\ , \quad i,j=0,\cdots,N-1\ . \tag{34}$$

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. (33) by $k_i\sqrt{w_i}$ to get:

$$\sum_j \widetilde{H}_{ij}\,[\widetilde{\psi}_n]_j = E_n\,[\widetilde{\psi}_n]_i\ , \tag{35}$$

where

$$[\widetilde{\psi}_n]_i \equiv k_i\sqrt{w_i}[\psi_n]_i\ . \tag{36}$$

This means that $\widetilde{H}_{ij}$ is

$$\widetilde{H}_{ij} \equiv \frac{k_i^2}{2\mu}\delta_{ij} + \frac{2}{\pi}k_i\sqrt{w_i}\,V(k_i,k_j)\,k_j\sqrt{w_j}\ , \quad i,j=0,\cdots,N-1\ , \tag{37}$$

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

$$[\widetilde{\psi}_n]\cdot[\widetilde{\psi}_n] = \sum_i [\widetilde{\psi}_n]_i^2 = 1\ , \tag{38}$$

then

$$1 = \sum_i k_i^2 w_i [\psi_n]_i^2 \longrightarrow \int_0^\infty k^2\, dk\ |\psi(k)|^2\ , \tag{39}$$

so the continuum version is normalized as well.

i. Delta-Shell Potential

A useful potential for testing is the "delta-shell" potential, which in the coordinate representation is

$$V(r) = \frac{\lambda}{2\mu}\delta(r-b) \; , \tag{40}$$

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. (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

$$E = -\frac{\kappa^2}{2\mu} \; , \tag{41}$$

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

$$e^{-2\kappa b} - 1 = \frac{2\kappa}{\lambda} \qquad (l = 0) \; . \tag{42}$$

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

$$1 - \frac{\lambda}{i\kappa}(i\kappa b)^2 j_l(i\kappa b)[n_l(i\kappa b) - i j_l(i\kappa b)] \; . \tag{43}$$

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

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

$$V(k',k) = \int_0^\infty r^2 dr \, j_l(k'r)\frac{\lambda}{2\mu}\delta(r-b) j_l(\kappa r) = \frac{\lambda b^2}{2\mu} j_l(k'b) j_l(kb) \; , \tag{44}$$

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

$$\psi_0(r) = \int_0^\infty k^2 dk \, \psi_0(k) j_0(kr) \propto \begin{cases} e^{-\kappa r} - e^{\kappa r} \; , & \text{for } r < b \; , \\ e^{-\kappa r} \; , & \text{for } r > b \; . \end{cases} \tag{45}$$

j. References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 780.20 info webpage for details on a new version.]

[2] M. Hjorth-Jensen, *Lecture Notes on Computational Physics* (2009). These are notes from a course offered at the University of Oslo. See the 780.20 webpage for links to excerpts.

[3] W. Press *et al.*, *Numerical Recipes in C++*, 3rd ed. (Cambridge, 2007). Chapters from the 2nd edition are available online from `http://www.nrbook.com/a/`. There are also Fortran versions.