

REVIEW: Poisson's Equation Solution

Poisson's Equation

Poisson's equation relates the potential function $V(x, y, z)$ to the charge density $\rho(x, y, z)$ enclosed in a particular volume

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\epsilon_0}$$

In the MKS units, $\epsilon = 8.854 \times 10^{-12}$ Coulomb²/Newton-meter². For a point charge q at the origin, and an infinite volume, then Poisson's equation has the Coulomb's law solution in spherical coordinates

$$V(r) = \frac{q}{4\pi\epsilon_0 r}$$

where the radial distance $r = \sqrt{x^2 + y^2 + z^2}$ in terms of the Cartesian coordinates.

As you might have anticipated, the Poisson equation in finite space can be solved with the relaxation methods that we have used for Laplace's equation. The only addition is that we add the charge density term to the iteration equation. The result, on page 144, is

$$V(i, j, k) = \frac{1}{6}[V(i+1, j, k) + V(i-1, j, k) + V(i, j+1, k) + V(i, j-1, k) + V(i, j, k+1) + V(i, j, k-1)] + \frac{\rho(x, y, z)\Delta x^2}{6\epsilon_0}$$

To simplify matters, the book works in units of charge such that $\epsilon_0 = 1$.

The simplest problem is for the point charge such that ρ is zero everywhere except at the origin. At the origin $\rho(0, 0, 0) = q/dx^3$ according to the textbook. As mentioned in the last class, the introduction of the differential dx is strange at this point since the iteration equation used Δx . The key point to remember is what is the total charge, namely the integral of ρ over the total volume where $\rho \neq 0$. In this problem we can compute the total charge as $q = \rho \times (\Delta x)^3$. So as long as we re-scale the charge density ρ according to the value Δx , then we will be dealing with the same amount of point charge. We will return to this discussion of charge density on the next page.

Example

In the last class the assignment was to study the program which solves Poisson's equation for a point charge by the SOR method using the example in Figure 5.8 of a finite size metal cube. You were also asked to develop a *help* improvement at the command line level.

For the Figure 5.8, the potential zero on the six walls of the cube, and the charge q/ϵ_0 is in the center of the box. You should get results such as shown in Figures 5.9 and 5.10, taking the box's origin at $(0, 0, 0)$ and having sizes which are first ± 2 units, and then ± 4 units. The grid size is 0.2 units.

In class today we will work with the *threeDPoisson.cpp* program. This program solves the point charge problem with either of the three methods we have learned: 1) Gauss-Jacobi, 2) Gauss-Seidel, or 3) Simultaneous Over-Relaxation (SOR).

We will also show how a *help* feature can be implemented at the command line.

Poisson Equation Point Charge Solution

Differential Equation

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\epsilon_0}$$

1) Gauss-Jacobi (G-J) algorithm

$$V_{\text{new}}(i, j, k) = \frac{1}{6} [V_{\text{old}}(i+1, j, k) + V_{\text{old}}(i-1, j, k) + V_{\text{old}}(i, j+1, k) + V_{\text{old}}(i, j-1, k) + V_{\text{old}}(i, j, k+1) + V_{\text{old}}(i, j, k-1)] + \frac{\rho(x, y, z)\Delta x^2}{6\epsilon_0}$$

2) Gauss-Seidel (G-S) algorithm The same iteration equation is used except that as soon as a $V_{\text{new}}(i, j, k)$ is available it is used in the right hand side of the equation.

3) Simultaneous-Over-Relaxation There are three steps. First the Gauss-Seidel solution is obtained for all grid points, and we can denote this solution as $V^{GS}(i, j, k)$. Then the difference

$$\Delta V(i, j, k) = V^{GS}(i, j, k) - V_{\text{old}}(i, j, k)$$

is computed. Lastly, an updated $V_{\text{new}}(i, j, k)$ array is computed as

$$V_{\text{new}}(i, j, k) = \alpha \Delta V(i, j, k) + V_{\text{old}}(i, j, k)$$

The α coefficient varies between 1 and 2. For a two-dimensional square array (which is not the Example 5.8) the best α coefficient can be proved to be

$$\alpha \approx \frac{2}{1 + \pi/L} \quad \text{where } L \text{ is the grid dimension.}$$

threeDPoisson Program

The *threeDPoissonProgram* has the capability to investigate all three solutions methods, according to the input command used. The program does this by making use of command line options, which is a feature of C/C++ as well as the Perl scripting language. In C/C++ the command line options feature is of the form

threeDPoisson [choice] [alphaCoeff]

where the *choice* value can be 1, 2, or 3 for the G-J, G-S, or SOR methods. If the SOR method is chosen, then a specific value of the α coefficient can be used. An important programming feature is that there are default values taken if no input option is given. So if no *choice* value is specified, then the G-J method is used. If no *alphaCoeff* value is specified, then a default value of 1.67 is used. Finally by typing the command line

threeDPoisson help

you will get a printout of the meaning of the input options, and their defaults if no input is given at the command line. We will study this program in class today.

Poisson Equation Point Charge Solution

Reason for renormalizing the charge density

One of the exercises we want to do is the calculate the potential function for different sized cubes, and for different numbers of grid points in that cube. In order to compare the different solutions properly, we want to make sure that we are looking at the same amount of point charge in each. The idea is that as we have a larger number of grid points then we should have a more accurate, converging numerical solution for the same box size and the same amount of point charge.

Technical solution for charge density renormalization

As states previously, the way in which we can insure that we are solving for the same amount of point charge is to keep the same total point charge value

$$q = \rho \times (\Delta x)^3$$

In the *threeDPoisson.cpp* program this is done by taking a default (reference) charge of $q = 1$ unit, a default $\Delta x_R = 0.2$ units, and a default box size of 4 units on each side. A default reference charge density ρ_R is computed as $\rho_R = q/(\Delta x_R)^3$. It is assumed that there will be a certain number of grid points L for which the box length will scale as $L\Delta x$.

For any other problem in which a different L , or a different $\Delta x'$ is taken which should be an integer multiple M of 0.2 units, then a scaled density is computed as

$$\rho_S = \rho_R * S \quad \text{where} \quad S \equiv \frac{(L-1)^3}{M^3}$$

and where L is the number of grid points being used. The number of grid points L is called *IROWS* in the program, and this number should be an odd number such as 7. In that way, the mid-point will be a well defined integer as $(IROWS + 1)/2$.

Finally, the $\Delta x'$ is scaled according to the number of grid points being used

$$\Delta x'_S = \Delta x_R \frac{M}{(L-1)}$$

In this manner, the inhomogeneous term in Equation 5.10 is automatically renormalized properly as

$$\frac{\rho_R(\Delta x_R)^2}{6} \longrightarrow \frac{\rho_S(\Delta x'_S)^2}{6}$$

Graphing the numerical result compared to the analytic solution

The analytic solution for a point charge q in infinite space is given by

$$V(r) + \frac{1}{4\pi\epsilon_0 r}$$

For the numerical solution in the graphical plot we take the origin to be at the midpoint of the box. By symmetry we only have to plot one quadrant of the box. We will look at plots for larger L , which in turn is effectively smaller Δx . We can figure out the distance from the origin using the Pythagorean theorem for points which are not along a given axis.

REVIEW: Numerical Integration

Biot-Savart Law

The magnetic field \vec{B} produced by a current carrying straight wire can be computed from the Biot-Savart law which gives the differential element $d\vec{B}$ in terms of the radial displacement \vec{r} from the wire, and the current element $I d\vec{z}$ in the wire

$$d\vec{B} = \frac{\mu_0 I}{4\pi} \frac{d\vec{z} \times \vec{r}}{r^3}$$

As you can see, this equation requires that a vector cross-product be computed. For a typical case, as in Figure 5.11 on page 149, the magnitude dB can be written as

$$dB = \frac{\mu_0 I}{4\pi} \frac{dz \sin \theta}{r^2}$$

The wire is in the z direction, and we are computing the magnetic field at a fixed distance x away from the mid-point $z = 0$ of the wire, for which $r^2 = x^2 + z^2$, and $\sin \theta = x/r$. The integration $B = \int dB$ can be carried out as a discrete sum

$$B \approx \frac{\mu_0 I}{4\pi} \sum_{i=1}^n \frac{x \Delta z}{(z_i^2 + x^2)^{3/2}} = \frac{\mu_0 I x \Delta z}{4\pi} \sum_{i=1}^n \frac{1}{(z_i^2 + x^2)^{3/2}}$$

In the above one divides up the length of the wire into n intervals with the appropriate z_i values, and then makes the sum.

Simpson's Rule

A more accurate approach than a simple sum at equidistant intervals is to use *Simpson's rule*. Simpson's rule is based on a parabolic approximation to the integrand $f(y)$ over a small interval. The final Simpson's Rule result (page 502) for n intervals is

$$\int_a^b f(y) dy \approx \frac{\Delta y}{3} [f(a) + 4f(y_1) + 2f(y_2) + 4f(y_3) + \dots + 2f(y_{n-2}) + 4f(y_{n-1}) + f(b)]$$

In the above formula n needs to be an even integer, that is, the coefficient of the y_i terms where i is odd is 4, and the coefficient of the y_i terms where i is even is 2. The individual $y_i = a + i\Delta y$, from which $\Delta y = (b - a)/n$

Numerical Integration for Biot-Savart Law

Example of a current carrying wire

The example is taken from exercise 5.11, comparing the simplest integration formula (Equation 5.25) with that of Simpson's rule for the same grid size Δz . Do two different grid sizes. For your calculations take a thin wire 1 meter long, with a current of 10 milli-amperes flowing (to keep it somewhat realistic). Calculate the magnetic field at $r = 5$ and $r = 15$ cm, requiring a convergence criterion between the two grid size of 1% in the result.

Compare your results to the infinite wire length Ampere's Law result

$$B(r) = \frac{\mu_0 I}{2\pi r}$$

where r is the radial distance away from the wire, and $\mu_0 = 4\pi \times 10^{-7}$ Tesla-meter/Ampere.

For this class, we will explore the use of the two programs *biotSavartSimple.cpp* and *biotSavart-Simpsons.cpp* to see how well our numerical and physics expectations are confirmed. Unlike the *threeDPoisson.cpp* example, these two programs are entirely *hard-coded*, meaning that there are not input options which can change the parameters of the calculations, Instead, one has to recompile anytime that there is a change. On the other hand, the programs are very simple so recompilation does not take long.

Waves on a String

Description of Wave Motion

We are all familiar with the motion of a transverse wave pulse on a taut string, or a slinky toy. This is motion in two dimensions (x, y) where the x direction is along the string. The y direction is transverse to the string and measures the amplitude of the wave. The motion is also time dependent. Thus the solution is of the form $y(x, t)$.

Transverse waves on a string have a differential equation of motion as follows:

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}$$

where c is a speed parameter. For a string under tension T and having a mass per unit length value μ the speed is given by $c = \sqrt{T/\mu}$.

At first sight, this equation seems to resemble the Laplace or Poisson equations which we solved using the relaxation method. However, that method is not applicable to the wave equation because we are seeking a time-dependent, not a stationary solution. However, as with the Laplace equation, we will consider the solution $y(x, t)$ to be a set of grid points $y(i, n)$. Similarly, we will impose a boundary condition by keeping the ends of the string fixed. Finally, we will need, as in any motion problem, an initial ($t = 0$) condition describing the string.

Numerical Solution to the Wave Equation

Just as in the Laplace equation, we can write a finite difference version of the wave equation which will lead to an integration equation for our numerical solution. In terms of the grid points approximation $y(i, n)$, this differential equation looks like

$$\frac{y(i, n+1) + y(i, n-1) - 2y(i, n)}{(\Delta t)^2} \approx c^2 \left[\frac{y(i+1, n) + y(i-1, n) - 2y(i, n)}{(\Delta x)^2} \right]$$

We can then write the iteration solution as

$$y(i, n+1) = 2[1 - r^2]y(i, n) - y(i, n-1) + r^2[y(i+1, n) + y(i-1, n)]$$

$$r \equiv \frac{c\Delta t}{\Delta x}$$

In the iteration on the left side we have the new time value $t = (n+1)\Delta t$, while on the right side we have the two previous time values $t = n\Delta t$ and $t = (n-1)\Delta t$. In order to start the iteration we must specify the condition of the string at all points for two time intervals. The usual approach is to say that the string has a specific say Gaussian pulse form for the two time intervals before the calculation starts.

$$y_0(x) = \exp[-k(x - x_0)^2]$$

The x_0 parameter gives the center of the Gaussian pulse while the k parameter gives the width of the pulse. As stated before, we need to impose boundary conditions on the string at the two ends. The simplest first choice is to have the ends fixed: $y(0, n) = y(M, n)$ for all times n , and where the length of the string $L = M\Delta x$.

Numerical Solution to Waves on a String

Example 6.1 in Figure 6.2

The pseudo-code for solving the wave equation is given in Example 6.1 on page 159, with the results shown in Figure 6.2. A C/C++ implementation of that pseudo-code is the *fixedString.cpp* program which we will study in this class. The most complicated part of this program is producing the cascade of snapshots of the string at different times, which can eventually be animated. We will explore the different possibilities of the r parameter as discussed on page 161.