27 January 2025 – PHYS 6260
Prof. John Wise
Newman Chapter 9: Partial Differential Equations

Some of the most interesting physics problems involve partial differential equations (PDEs), such as the wave equation, diffusion equations, the Laplace and Poisson equations, Maxwell's equations, and the Schrödinger equation. The method to solve PDEs is straightforward but computationally intense because of the multi-dimensional aspect of them. We'll first cover boundary value problems because they're easier to address and then move to initial value problems.

Boundary value problems with PDEs have some specified constraint on the boundaries that need to be enforce all while the interior points conform to a governing set of PDEs. One example that we will first inspect is an electrostatics problem.

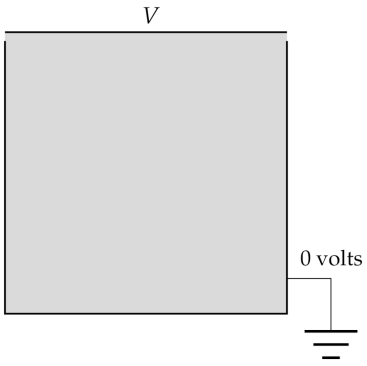

Figure 1: **A simple electrostatics problem.** An empty box has conducting walls, all of which are grounded at 0 V expect for the top wall that has a potential of V volts.

Consider an empty conducting box, illustrated in Figure 1, that has a voltage V applied to the top wall and the other five walls are grounded. What is the electric potential inside the box? Recall from your E&M class that the electric field is related to its potential through $\vec{E} = -\nabla\phi$ and Gauss's Law gives us $\nabla \cdot \mathbf{E} = 0$ for this situation. Thus, we have Laplace's equation, which is the partial differential equation,

$$\nabla^2\phi = 0 \tag{1}$$

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = 0 \tag{2}$$

Since it is an electrostatic problem, it is time-independent, which makes our lives a lot easier.

For initial value problems, the solution is time varying, which makes the solution much harder. An example of an initial value problem is a string being plucked, and we are tasked to solve for the motion of the string, given some initial impulse and it being fixed at both ends. We will focus on these problems in the next lecture.

9.1: Relaxation Method: Boundary Value Problems

A fundamental technique for solving PDEs is the *method of finite differences*. We will be inspecting two-dimensional problems in class because it demonstrates the multi-dimensional aspect of these problems. It is relatively straightforward to extend these prescriptions to three (and higher) dimensions. Let's consider the electrostatics problem set out earlier but

in two dimensions. We have Laplace's equation,

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \tag{3}$$

which we want to solve for the electric potential ϕ given the boundary constraints. We can construct various types of sets of discrete points in which we calculate $\phi(x, y)$. However, the most common and easier type is a regular Cartesian grid with points equally spaced by a. Consider a single point at (x, y), we've learned in the past how to numerically express second derivatives, and we can apply this to the equation above.

$$\frac{\partial^2\phi}{\partial x^2} = \frac{1}{a^2}\left[\phi(x+a, y) + \phi(x-a, y) - 2\phi(x, y)\right] \tag{4}$$

$$\frac{\partial^2\phi}{\partial y^2} = \frac{1}{a^2}\left[\phi(x, y+a) + \phi(x, y-a) - 2\phi(x, y)\right] \tag{5}$$

We can insert these two expressions into Equation (3) and remove the $1/a^2$ factor to obtain

$$\phi(x+a, y) + \phi(x-a, y) + \phi(x, y+a) + \phi(x, y-a) - 4\phi(x, y) = 0. \tag{6}$$

This is just a linear equations that we need to solve at each point, which is much easier than solving a PDE. Solving for the potential at point (x, y), we find

$$\phi(x, y) = \tfrac{1}{4}\left[\phi(x+a, y) + \phi(x-a, y) + \phi(x, y+a) + \phi(x, y-a)\right], \tag{7}$$

which is just average of the neighboring points. This is called the *Jacobi method*.

In principle, this is N^2 equations with N^2 unknowns, which could make for a difficult numerical solution using Gaussian elimination or LU decomposition. However, we can use the relaxation method to solve for these values. Here we fix the values at the boundaries and take the average of the neighboring points from the previous solution for each point. Let's call this updated solution $\phi'(x, y)$. Given this updated solution, we can then repeat this process, and over several iterations, the solution relaxes to some solution. We stop the iteration when the change in the solution, $\phi'(x, y) - \phi(x, y)$ is less than some target accuracy at *every point*.

The relaxation method with PDEs can suffer from the same problems as we saw in solving non-linear equations, in which the solution diverges from the actual solution. However the Jacobi method does not have this problem, and it can be proved that it is always stable.

Example 9.1: Solution of Laplace's Equation

Let's solve Laplace's equation for the example electrostatics problem that we were previously discussing. A voltage of $V = 1$ V is applied to the top wall. The box has a side length of 1 m, and we want to calculate ϕ at a spacing of 1 cm, resulting in 101^2 points, including the boundaries. Here is a program that uses the relaxation method for this system and makes a density plot of the potential.

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
M = 100         # Grid squares on a side
V = 1.0         # Voltage at top wall
target = 3e-4   # Target accuracy

# Create arrays to hold potential values
phi = np.zeros([M+1,M+1],float)
phi[0,:] = V
phiprime = np.empty([M+1,M+1],float)

# Main loop
delta = 1.0
iteration = 0
while delta>target:

    # Calculate new values of the potential
    for i in range(M+1):
        for j in range(M+1):
            #
            # Complete in-class for phiprime, the updated solution
            # Remember to keep the boundaries at zero
            #

    # Calculate maximum difference from old values
    delta = np.max(abs(phi-phiprime))
    if iteration % 10 == 0:
        print("Iteration %3d: max. residual = %12.6g" % (iteration, delta))

    # Swap the two arrays around
    phi,phiprime = phiprime,phi
    iteration += 1

# Make a plot
plt.imshow(phi)
plt.gray()
plt.show()
```

This program results in the image shown in Figure 2. You will notice that the system size L or the spacing a never enter into the solution because the relaxation method only calculates the averages of the adjacent values. Additionally, this solution takes about 30 seconds, mainly from the loops. In class, we will try to vectorize the math operations to

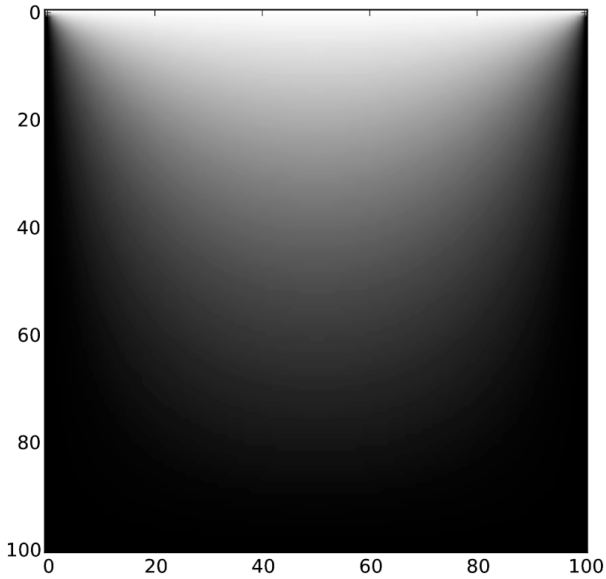

Figure 2: **Solution of the Laplace equation** in the electrostatic problem given in Example 9.1. The electric potential is depicted in a density plot.

accelerate the computation.

One can see that it took over 780 iterations for the relaxation problem to converge to a solution. This mainly comes from the second derivative being only first-order accurate. We can use higher-order approximations to make the solution converge faster, or we could use more grid points, resulting in a more accurate solution but slower calculation.

Another point to notice is that boundaries that don't align with Cartesian axes are not well described by a regular Cartesian grid, for example, diagonal, circular, or any type of irregular boundary. The grid points don't exactly fall on a set of grid points, and we would have to approximate the boundary to the closest point. Other methods would have deformable or irregularly-shaped grids or adaptive grids.

Example 9.2: Solution of the Poisson Equation

Let's look a little more complex electrostatic problem, where we have a grounded conducting hollow 2-D box with a side length of 1 m but with two charged areas. We would use the Poisson equation to solve for the potential,

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0}, \tag{8}$$

where ϵ_0 is the permittivity of empty space. The two charged areas have a surface charge density of ± 1 C m^{-2} and have a side length of 20 cm. Their lower-left corners are located at $(x, y) = (0.2, 0.2)$ m and $(0.6, 0.6)$ m. The system is shown in Figure 3.

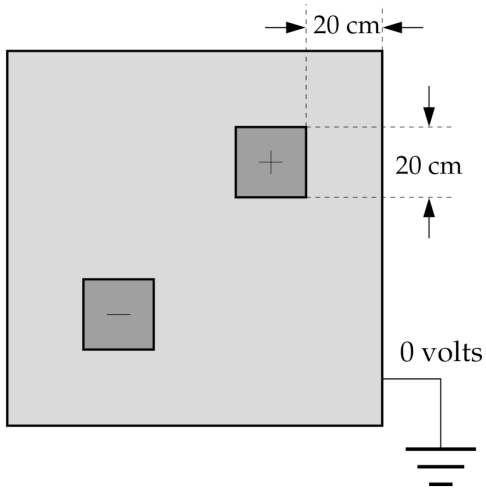

Figure 3: **A more complicated electrostatics problem** with two square charged inside a square 2-D box. The box is grounded so that the potential at the boundaries of the box is zero.

We use the finite differencing of the Laplacian again to express the Poisson equation as

$$\frac{1}{a^2}\left[\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)-4\phi(x,y)\right]=-\frac{\rho(x,y)}{\epsilon_0}. \qquad (9)$$

Solving for $\phi(x,y)$, we find

$$\phi(x,y)=\frac{1}{4}\left[\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right]+\frac{a^2}{4\epsilon_0}\rho(x,y). \qquad (10)$$

The last term is known as the *source / sink term* that supplies the charge density for the potential. We can still use the relaxation method to calculate the potential but the average is supplemented with the source term.

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
M = 25          # Grid squares on a side
V = 0.0          # Voltage at top wall
L = 1.0          # Side length of box
target = 3e-4    # Target accuracy
epsilon_0 = 8.85e-12
a = L / float(M) # grid point spacing

# Create arrays to hold potential values
phi = np.zeros([M+1,M+1],float)
phi[0,:] = V
```

```
phiprime = np.empty([M+1,M+1],float)

# Create surface charge density array
rho_values = [-1.0, 1.0]  # Surface charge density
charge_width = [0.2, 0.2] # Width of the charges (assuming squares)
charge_boundaries = [[0.2, 0.2], [0.6, 0.6]] # Position of origin of the charges
rho = np.zeros([M+1,M+1], float)
for i in range(len(rho_values)):
    #
    # Complete in-class. Define rho (source/sink term)
    # Fill in the charged squares with the appropriate density
    #

# Main loop
delta = 1.0
iteration = 0
while delta>target:

    # Calculate new values of the potential
    for i in range(M+1):
        for j in range(M+1):
            #
            # Complete in-class for phiprime, the updated solution
            # Remember to keep the boundaries at zero and
            # to include the sink/source term
            #

    # Calculate maximum fractional difference from old values
    delta = np.abs(phi-phiprime)
    nonzero = phi > 0
    if nonzero.any():
        delta = np.max(delta[nonzero] / phi[nonzero])
    else:
        delta = 1.0
    if iteration % 10 == 0:
        print("Iteration %3d: max. residual = %12.6g" % (iteration, delta))

    # Swap the two arrays around
    phi,phiprime = phiprime,phi
    iteration += 1

# Make a plot
plt.imshow(phi)
plt.gray()
plt.show()
```

9.2: Faster Methods for Boundary Value Problems

The Jacobi method is straightforward and simple to program, however it takes a long time to converge to a solution. We will cover two primary methods to overcome this shortcoming.

Over-relaxation

Suppose that the relaxation method is steadily marching toward some solution. But we want to accelerate this progression. For instance, the solution at some grid point might be 0.5, whereas our initial guess is 0.1. In the first iteration, it goes from 0.1 to 0.3. But we can say, let's overshoot this estimate a little and go to 0.4. In practice, we don't know the actual solution, but the method of *over-relaxation* will get us there with less iterations.

Consider again the Laplace equation in the first example in class. We have a current solution $\phi(x, y)$ and an updated solution $\phi'(x, y)$, which we can write as

$$\phi'(x, y) = \phi(x, y) + \Delta\phi(x, y), \tag{11}$$

where $\Delta\phi$ is the change in ϕ in this step. Now we define a set of over-relaxed values

$$\phi_\omega(x, y) = \phi(x, y) + (1 + \omega)\Delta\phi(x, y), \tag{12}$$

where $\omega > 0$, meaning that we change the solution by a little more than we normally would in a relaxation scheme. Now we can substitute $\Delta\phi$ into the previous equation to obtain

$$\begin{aligned}
\phi_\omega(x, y) &= \phi(x, y) + (1 + \omega)[\phi'(x, y) - \phi(x, y)] && (13)\\
&= (1 + \omega)\phi'(x, y) - \omega\phi(x, y) && (14)\\
&= \frac{1 + \omega}{4} [\phi(x + a, y) + \phi(x - a, y) + \phi(x, y + a) + \phi(x, y - a)] - \omega\phi(x, y) && (15)
\end{aligned}$$

The over-relaxation method is thus very similar as the Jacobi method, but we have to choose some value for ω that results in a similar solution. We'll come back to this detail later.

The Gauss-Seidel Method

The second numerical trick to speed up the Jacobi method is called the *Gauss-Seidel method* in which we use the new values in the averaging as they are calculated. These new values are, in principle, closer to the solution and by using them, it will accelerate the convergence. This means that once we have calculated the new solution at some grid point, we can delete the old solution. A practical way to accomplish this technique is using only a single array for the field. It will contain both new and old solutions. We can use computer science notation by stating

$$\phi(x, y) \leftarrow \tfrac{1}{4} [\phi(x + a, y) + \phi(x - a, y) + \phi(x, y + a) + \phi(x, y - a)] , \tag{16}$$

that denotes the right-hand side replaces the left-hand side value. We can combine the Gauss-Seidel and over-relaxation methods that will converge even faster, which can be encapsulated as:

$$\phi(x,y) \leftarrow \frac{1+\omega}{4}\left[\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right]-\omega\phi(x,y), \quad (17)$$

Another point to note is that the over-relaxation method alone can be *unstable* in that the estimated solution will diverge from the actual one. However, for reasons not discussed in this class, the Gauss-Seidel method paired with over-relaxation is formally *stable* and will never diverge.

Choosing the over-relaxation parameter

There is no exact method to choose an optimal over-relaxation parameter ω. It really depends on the specific equations that we are solving, the boundary conditions, and the grid properties (spacing, shape, etc). There are some guidelines, though. The larger the ω value, the faster the calculation will proceed but only to a point. If the ω value is too large, then it will become unstable, possibly oscillating around the solution. However, the method is generally stable for $\omega < 1$. In practice, the optimal value is usually determined for a set of equations and grid type through experimentation. For instance, when solving the Laplace equation, $\omega = 0.9$ works best and accelerates the calculation tremendously (by 10–20 times).

9.3: Initial value problems

Boundary value problems are only one type of PDE, which are time independent. There are many systems that are time-independent that evolve with time. These are called **initial value problems**, and can have boundary conditions, too. Our goal is to predict their future variation as a function of time. A simple example is the diffusion (heat) equation in one dimension:

$$\frac{\partial\phi}{\partial t} = D\frac{\partial^2\phi}{\partial x^2}, \quad (18)$$

where D is the diffusion coefficient. This equation is used to calculate the motion of diffusing gases and liquids, as well as the flow of heat in thermal conductors.

The variable $\phi(x,t)$ depends on space and time. One can think about this problem as a two-dimensional problem, just like how we previously solved the 2D Laplace equation for boundary value problems. Now we can consider a "space-time grid", where space and time are discretized, and we write the above PDE in finite difference form.

We have an initial system setup, and we need to predict how it behaves afterward. Thus, we do not know the endpoint of the system, so we can't use a relaxation method. The only methods that are feasible are based on forward integration.

The FTCS Method

Consider the solution of the diffusion equation (Equation 18). Just like how we divided the spatial domain into discrete points in multi-dimensional boundary condition problems, we can do the same thing in the temporal dimension. First we will use evenly split points, so we can write the right-hand side as

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{1}{a^2}\left[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)\right] \tag{19}$$

giving

$$\frac{\partial \phi}{\partial t} = \frac{D}{a^2}\left[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)\right] \tag{20}$$

If we can think about the variables at different grid points as separate variables, we now have a set of simultaneous ODEs, and we have learned how to solve them. On a single compute core, it is no problem to solve hundreds, thousands, and up to a few million of them.

We have choose a method to solve the left-hand side of Equation (20). We can notice that the right-hand side is only first-order accurate with second-order errors. So there's little benefit in representing the left-hand side with a higher-order method. So we can choose Euler's method to advance the system forward in time, even though this seems like a strange choice, given that Runge-Kutta methods are not much harder to implement. However its higher accuracy would be wasted by the second-order error from the second derivative.

Using Euler's method, we can write the value of ϕ at some future time, $t + h$,

$$\begin{aligned} \phi(t+h) &\simeq \phi(t) + h\frac{d\phi}{dt} \\ &\simeq \phi(x,t) + h\frac{D}{a^2}\left[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)\right] \end{aligned} \tag{21}$$

Thus, if we know the value of ϕ at every point at some time t, we can calculate their values at the future time $t+h$. This is very similar to the solution of ODEs in the previous chapter, but now we are mixing both time and spatial derivatives. This is called the **forward-time, centered-space (FTCS)** method for solving PDEs.

Example 9.3: The Heat Equation

The flat base of a container made of 1 cm thick stainless steel is initially at a uniform temperature of 20° Celsius everywhere. The container is placed in a bath of cold water at 0°C and filled with hot water at 50°C.

Our goal is to calculate the temperature profile of the steel as a function of distance x from the hot side to the cold side of the container, and as a function of time. For simplicity, let's consider the base infinitely wide, so that the temperature profile doesn't depend on position.

Thermal conduction is governed by the diffusion equation, so solving this problem is a matter of solving the one-dimensional diffusion equation for the temperature T:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \tag{22}$$

Let's divide the x-axis into 100 equal grid intervals, meaning that there will be 101 grid points in total counting the first and last ones. The first and last points have fixed temperature of 50° and 0°C, respectively, while the intermediate points are initial all at 20°C. We also need the heat diffusion coefficient for stainless steel, which is $D = 4.25 \times 10^{-6}$ m^2 s^{-1}.

Here is a program to perform the calculation using the FTCS method, making a temperature profile plot at $t = (0.01, 0.1, 0.4, 1, and 10)$ s, all on the same graph.

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
L = 0.01      # Thickness of steel in meters
D = 4.25e-6   # Thermal diffusivity
N = 100       # Number of divisions in grid
a = L/N       # Grid spacing
h = 1e-4      # Time-step
epsilon = h/1000

Tlo = 0.0     # Low temperature in Celcius
Tmid = 20.0   # Intermediate temperature in Celcius
Thi = 50.0    # Hi temperature in Celcius

t1 = 0.01
t2 = 0.1
t3 = 0.4
t4 = 1.0
t5 = 10.0
tend = t5 + epsilon

# Create arrays
T = np.empty(N+1,float)
T[0] = Thi
T[N] = Tlo
T[1:N] = Tmid
Tp = np.empty(N+1,float)
Tp[0] = Thi
Tp[N] = Tlo
```

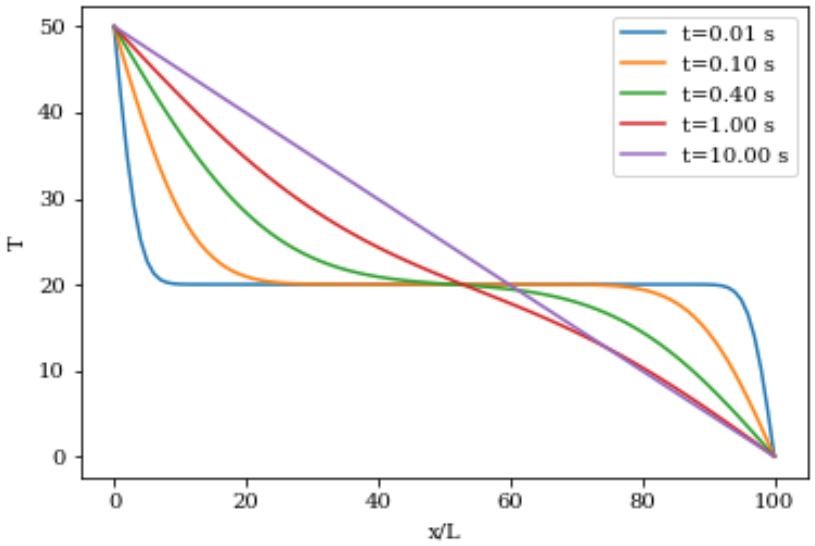

Figure 4: Evolution of temperature profile in Example 9.3

```
# Main loop
t = 0.0
c = h*D/(a*a)
while t<tend:
    # In-class problem
    #
    # Calculate the new values of T

    # In-class problem
    #
    # Make plots at the given times (t1 .. t5)


plt.xlabel("x")
plt.ylabel("T")
plt.show()
```

In this example, we assumed that the boundaries are time-independent, but in reality, the hot and cold water bath temperatures would have been changing. The FTCS method would remain the same, but the boundary conditions would have been changing with time, which would be altered before evaluating the FTCS portion.