# 12 Oct 23 - Activity: Method of Relaxation

As we discussed, we can use the relaxation approach to solve Laplace’s equation. As a reminder the process for doing this is as follows:

1. Divide the region into a grid
2. Set the boundary conditions on the edges of the grid
3. Guess the starting values for the potential in the interior of the grid (not on the edges)
4. Calculate the potential at each point in the grid using the average of the neighboring points (replacing the value as you go)
5. Repeat step 4 until the values in the grid stops changing

Below, we have written up a 1D code that performs each of these steps. Your task is to modify this code to work in 2D. You will need to write the functions that compute the potential at each point in the 2D grid as well as the function that sets the boundary conditions.

import numpy as np  
import matplotlib.pyplot as plt

## 1 Dimensional Relaxation

The 1D problem is rather simple in that it produces a linear function – always. This is because the only solution to the 1D Laplace’s equation is a linear function.

The solution to this equation is . We can use this to test our relaxation code and make sure that it is accurate. In general, you cannot do this and instead need to have some other mechanism for checking the accuracy of your code (like [convergence](https://en.wikipedia.org/wiki/Rate_of_convergence)).

Below, we have written the 1D code as a set of functions.

### Initialize the Grid and set the Boundary Conditions

We wrote two functions initialize\_grid and set\_boundary\_conditions that initialize a linear grid and also set the boundary conditions. The initialize\_grid function simply sets all values to zero and then set\_boundary\_conditions sets the values at the edges to the correct values.

def initialize\_grid(size):  
 ## Set the initial values of the grid  
   
 return np.zeros(size)  
  
def set\_boundary\_conditions(grid, left\_val, right\_val):  
 ## Set the boundary conditions  
 grid[0] = left\_val  
 grid[-1] = right\_val  
   
 return grid

### Relaxation

The core relaxation function is run\_relaxation. Notice that it takes in the number of iterations (num\_iterations); that is, the number of times you want the relaxation algorithm to run. That is, it will calculate the average of the neighboring points fully, and repeat it a limited number of times.

**This is not how the typical relaxation approach works.**

Normally, you would run the algorithm until the changes between each iteration are below some error tolerance you’ve chosen. This can lead to (near)infinite loops if your approach: (1) doesn’t converge or (2) converges very slowly. So it is important to have a maximum number of iterations to run. **You will modify the 1D code to do this.**

Notice that our relaxation function stores the values of the potential at set iterations (store\_freq). This is to allow us to plot the potential at different points in the relaxation process. But we don’t want to keep them all as that can become memory intensive.

def run\_relaxation(grid, num\_iterations, store\_freq=500):  
 ## Run the relaxation algorithm  
 ## Store the grid at regular intervals  
 stored\_iterations = []  
   
 ## Run the relaxation  
 for i in range(num\_iterations):  
 for j in range(1, len(grid)-1):  
 ## Update the grid point with the average of the neighboring points  
 grid[j] = 0.5 \* (grid[j-1] + grid[j+1])  
   
 ## Store the grid every store\_freq iterations  
 if i % store\_freq == 0:  
 stored\_iterations.append(np.copy(grid))  
   
 return stored\_iterations

### Plotting

Our relaxtion algorithm stores a list of grids at different points in the relaxation process. We can use this to plot the potential at different points. Below, we have a function that does just that.

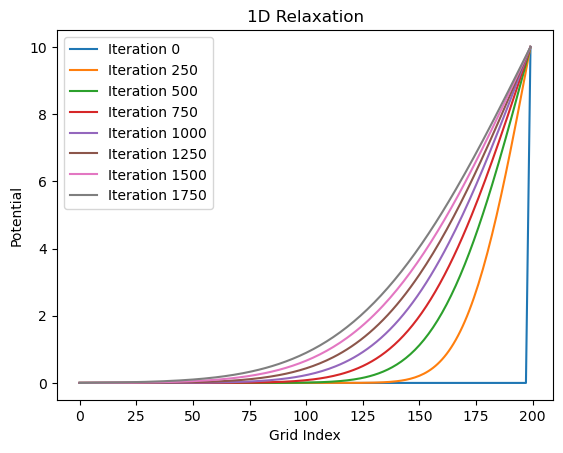
def plot\_results(stored\_iterations):  
 ## Plot the results (including all stored iterations)  
 for i, grid in enumerate(stored\_iterations):  
 plt.plot(grid, label=f"Iteration {i\*store\_freq}")  
 plt.legend()  
 plt.xlabel('Grid Index')  
 plt.ylabel('Potential')  
 plt.title('1D Relaxation')  
 plt.show()

### Let’s run it

We run the code below for linear grid of 100 points. The left boundary condition () is set to zero and the right one () is set to 10. Notice that there is an exact solution:

We can compare to that exact solution to see how well our relaxation algorithm is working. We will do that in a moment. First, let’s run the code.

# Parameters  
a = 1  
dx = 0.005  
grid\_size = int(1/dx)  
left\_bc = 0  
right\_bc = 10  
num\_iterations = 2000  
store\_freq = 250  
  
# Execution  
grid = initialize\_grid(grid\_size)  
grid = set\_boundary\_conditions(grid, left\_bc, right\_bc)  
stored\_iterations = run\_relaxation(grid, num\_iterations, store\_freq)  
plot\_results(stored\_iterations)



png

It definitely looks like it gets close to the exact solution. But how close? Let’s plot the difference between the exact solution and the solution from our relaxation algorithm. The code below computes the exact solution and then plots the absolute value of the difference between the two at each point on the grid.

def compute\_exact\_solution(grid, left\_bc, right\_bc):  
 x = np.linspace(0, len(grid)-1, len(grid))  
 return left\_bc + x \* (right\_bc - left\_bc) / (len(grid)-1)  
  
def plot\_error(stored\_iterations, exact\_solution):  
   
 # Plotting error  
 final\_result = stored\_iterations[-1]  
 error = np.abs(final\_result - exact\_solution)  
 plt.plot(error, label="Error", color='red')  
 plt.legend()  
 plt.xlabel('Grid Index')  
 plt.ylabel('Error')  
 plt.title('Error in 1D Relaxation')  
 plt.show()

exact\_solution = compute\_exact\_solution(grid, left\_bc, right\_bc)  
plot\_error(stored\_iterations, exact\_solution)



png

**✅ Do this**

1. Make sure you can run the code above and you have conceptual understanding of what it is doing. Test it and make sure it follows your intuition.
2. Modify the code to do a better job of predicting the exact solution. Get the maximum absolute error below 0.01, 0.001, 1e-4, 1e-5. How many iterations does it take to get to each of these values?
3. Modify the code to stop when the maximum absolute error is below a value of your choosing. Did you include a max\_iterations variable? If not, do so to avoid infinite loops.

## 2D Relaxation

The 2D relaxation code is very similar to the 1D code. One difference is that the grid has to be traversed in both directions. This means that you will need to move systematically through the 2D grid. The second is that you have to ensure the boundary conditions are set across the entire edge of the grid.

We recommend breaking the approach into similar functions as above.

1. Initialize the 2D grid of a chosen size (we’ve done that for you)
2. Set the boundary conditions (here you want a functions that can set each edge to a value or array of values) *Hint: try using cases or if else to send edge information (‘top’, ‘left’) and then use the : with the correct index to set the values for a full row or column (e.g., V[:,0])*
3. Relax the grid (here you will need to loop through the grid in both directions)

We’ve started the code for you below. You will need to finish it.

def initialize\_grid\_2d(size):  
 '''Initialize the grid of potential values'''  
 return np.zeros((size, size))  
  
def set\_boundary(phi, edge, value):  
 '''phi is the whole grid of potential values that you initialized  
 edge is the edge of the grid that you want to set (top, bottom, left, right)  
 value is the value you want to set the edge to (single number or array)'''  
  
def relax(phi, N, tolerance, max\_iterations=10000, store\_frequency=1000):  
 '''phi is the grid of potential values that you initialized  
 after you set the boundary conditions. That is key!  
 N is the size of the grid (assumed square  
 tolerance is the stopping criterion - you control this; be careful  
 max\_iterations is the maximum number of iterations to run (stops loop if not converged)  
 store\_frequency is how often you want to store the phi values (for plotting)'''  
   
 iterations = 0 ## Keep track of the number of iterations  
 delta = 1.0 ## Initialize delta (error) to be larger than tolerance  
   
 stored\_phi = [] ## Keep track of phi values for plotting  
 stored\_deltas = [] ## Keep track of delta values for plotting  
   
 ## Loop condition to run until convergence or max\_iterations  
 while delta > tolerance and iterations < max\_iterations:  
 # Store the old phi values to calculate delta later  
 phi\_old = phi.copy()  
   
 ######################  
 ### YOUR CODE HERE ###  
 ######################  
   
 # Calculate delta: max difference between new and old phi values  
 delta = np.max(np.abs(phi - phi\_old))  
   
 ## Increment the iteration counter  
 iterations += 1  
   
 ## Store phi and delta values every store\_frequency iterations  
 if iterations % store\_frequency == 0:  
 stored\_phi.append(phi.copy())  
 stored\_deltas.append(delta)  
 print(f"Iteration: {iterations}, Delta: {delta}")  
   
 return stored\_phi, stored\_deltas, iterations

**✅ Do this**

Finish the 2D relaxation code (below we’ve written up an example use case that it should work for) 1. You need to make sure the boundary conditions are set correctly 2. You need to traverse the grid and calculate the average of the neighboring points

It should be that once you have that working, the code below will run. You will need to plot it to see if it is working correctly.

# Example Usage  
L = 1.0  
h = 0.01  
tolerance = 1e-5  
N = int(L/h)  
  
phi = initialize\_grid\_2d(N+1)  
  
# Set varying potential on top boundary  
x = np.linspace(0, L, N+1)  
V\_top = 10 \* np.sin(2 \* np.pi \* x)  
set\_boundary(phi, 'top', V\_top)  
  
# Set constant potential on other boundaries  
set\_boundary(phi, 'bottom', 0)  
set\_boundary(phi, 'left', 5)  
set\_boundary(phi, 'right', 5)  
  
# Relaxation  
stored\_phi, stored\_deltas, iterations = relax(phi, N, tolerance)  
print(f"Converged in {iterations} iterations.")

Converged in 1 iterations.

## What about Poisson’s equation?

It turns out we can use a very similar approach when we have charges! Alia has written up a script that implements the relaxation approach for Poisson’s equation. You can find it below along with the derivation.

Where is a **charge density**.

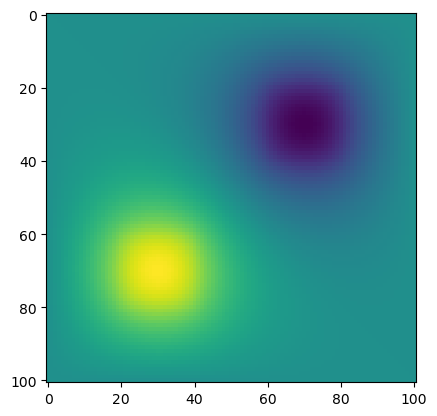
Substituting our 2nd partial derivative equations gives us:

Which in turn gives us the update equation:

We can implement this (warning: this takes a long time to converge).

# constants and parameters  
L = 1  
h = 0.01  
V = 1  
N = int(L/0.01)  
tol = 1e-2  
epsilon0 = 8.854e-12  
  
# setup phi, phiprime arrays  
phi = np.zeros((N+1,N+1))  
phiprime = np.copy(phi) # to play the part of phi (k+1)  
  
def rho(i,j):  
 x = i \* h  
 y = j \* h  
 if x >= 0.6 and x <= 0.8 and y >= 0.2 and y <= 0.4:  
 return 1e-7  
 elif x >= 0.2 and x <= 0.4 and y >= 0.6 and y <= 0.8:  
 return -1e-7  
 else:  
 return 0.  
  
k = 0 # track number of iterations  
delta = 1.0 # initial delta  
  
while delta > tol: # run until converged  
  
 for i in range(N+1):  
 for j in range(N+1):  
 if i == 0 or i == N or j == 0 or j == N: # don't update boundary  
 phiprime[i,j] = phi[i,j]  
 else:  
 # relaxation update equation  
 phiprime[i,j] = 0.25 \* (phi[i+1,j] + phi[i-1,j] + phi[i,j+1] + phi[i,j-1]) + h\*\*2 / (4 \* epsilon0) \* rho(i,j)  
  
 # find max difference for delta  
 delta = np.abs(phi-phiprime).max()  
  
 # swap arrays to keep iterating  
 phi,phiprime = phiprime,phi   
   
 # track iterations  
 k += 1  
 if k%500 == 0 or k ==1:  
 print("iter:",k,"delta:",delta)   
 delta = np.abs(phi-phiprime).max()  
  
print(k)  
plt.imshow(phi)  
plt.show()

iter: 1 delta: 0.28235825615541005  
iter: 500 delta: 0.06832141135411973  
iter: 1000 delta: 0.03251014051510026  
iter: 1500 delta: 0.017114857647044346  
1931



png