Simulations in Physics

Apurv Keer
Mridul Goel

§ WEEK 2 §

Problem 1: Solving system of linear equations numerically

Resource:

- https://en.wikipedia.org/wiki/Jacobi_method
- https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method

Solve the following system of linear equations using the Jacobi and Gauss-Seidel methods.

$$4x - y - z = 3 \tag{1.1}$$
$$-2x + 6y + z = 9 \tag{1.2}$$
$$-x + y + 7z = -6 \tag{1.3}$$

Choose the initial guess $\mathbf{x}^{(0)} = (0, 0, 0)$. For convergence use a maximum of 100 iterations or a tolerance of 10^{-5} i.e. $|\mathbf{x}^{(i+1)} - \mathbf{x}^{(i)}| < 10^{-5}$.

Problem 2: Solving Poisson's Equation Numerically using the Finite-Difference Method

Resources:

- https://gnuplotting.org/tag/colormap/

Solve 2D Poisson's equation for various charge configurations and make an cmap plot.

Theory The intuition behind the finite-difference method is the following:
Let's say you have some sort of linear differential equation involving a function and its derivatives. You know that you can make approximations of a derivative by considering the value of the function at a point and another point nearby.
When you write down the derivatives of $f(x)$ using the neighboring points such as $f(x+h), f(x-h)$, AND if the differential equation is linear, your differential equation reduces to the problem of solving a large set of linear equations, which have the variables $v_i = f(p_i = x + i * h), i \in \mathbb{Z}$ and p_i within the boundary we are trying to solve the equation in.
In fact, at the boundary, there should be known values $f(p_{boundary})$, that are also part of the set of linear equations for points near the boundary.
After all of this, the set of linear equations we are left with should hopefully have a solution, which is an approximate solution to the original differential equation.
Let's take Poisson's equation to be in 2 dimensions:

$$\frac{\partial^2 U(x,y,z)}{\partial x^2} + \frac{\partial^2 U(x,y,z)}{\partial y^2} = -\rho(x,y) \tag{2.1}$$

We know that the partial derivative w.r.t. a variable of a multivariate function (i.e. a function dependent on multiple variables) is simply the change in the function w.r.t. change in *only* this variable.

We also know that the approximation to the derivative of a function in 1D is:

$$f'(x) = \frac{f(x + h/2) - f(x - h/2)}{h} \tag{2.2}$$

We can derive an approximation to the second derivative using the approximation for the first derivative:

$$f''(x) = \frac{f'(x + h/2) - f'(x - h/2)}{h} \tag{2.3}$$

$$= \frac{(\frac{f(x + h/2 + h/2) - f(x + h/2 - h/2)}{h}) - (\frac{f(x - h/2 + h/2) - f(x - h/2 - h/2)}{h})}{h} \tag{2.4}$$

$$= \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} \tag{2.5}$$

Let us now do something called *discretization*.
Discretization simply takes a continuous function, variable, or anything, and makes it into discrete values. We cannot numerically solve Poisson's Equation for each coordinate point. Instead, we will solve Poisson's equation for a discrete *grid* of points; if we take a grid that is fine-grained enough (i.e. the value of h is sufficiently small), then we can be confident that the numerical solutions are close to what would happen in a real experiment.
Let us assign the indices (i, j) to the (discretized) coordinate $(x = i * h, y = j * h)$, where i and j are integers.
Thus we can also denote $U(x = i * h, y = j * h)$ as U_{ij}. Also denote h as Δi. In fact, if the coordinate grid is not of the same scale in the two axes i.e. $(x, y) = (i * h, j * k)$, then we write $h, k = \Delta i, \Delta j$.
For the charge distribution ρ, we can discretize it by finding the average charge within a grid cell and taking that to be the value of $\rho_{i,j}$.
Now, since the partial derivative of a function is essentially a 1D derivative over a particular variable, with all other variables appearing to be constant, we can write:

$$\frac{\partial^2 U(x, y)}{\partial x^2} = \frac{U(x + h, y) - 2U(x, y) + U(x - h, y)}{h^2} \tag{2.6}$$

$$= \frac{1}{\Delta i \Delta j}(U_{i+1,j} - 2U_{i,j} + U_{i-1,j}) \tag{2.7}$$

Similarly,

$$\frac{\partial^2 U(x, y)}{\partial y^2} = \frac{1}{\Delta i \Delta j}(U_{i,j+1} - 2U_{i,j} + U_{i,j-1}) \tag{2.8}$$

Plugging these results into Poisson's equation gives us:

$$U_{i+1,j} + U_{i,j+1} + U_{i-1,j} + U_{i,j-1} - 4U_{i,j} = -(\Delta i \Delta j)\rho_{i,j} \tag{2.9}$$

As you can see, this is an equation for $U_{i,j}$ given in terms of its neighboring points.
Note, that all 'Solving' Poisson's equation means is to find the function U in terms of the coordinates x and y, where U is subject to some charge distributions. In the discretized form, the boundary condition corresponds to known values of $U_{i',j'}$ where i', j' correspond to coordinates that "lie on the boundary", and in our case these are taken to have zero potential at the boundary. Thus, the discretized form of Poisson's equation and the boundary conditions form a set of Linear equations with the variables $U_{i,j}$ for all possible values of i and j. If we take our coordinate space to be a 10×10 coordinate grid, we will have i, j range from 0 to 9.

Implementation We write down the collection of all variables $U_{i,j}$ as a single vector by flattening the U matrix into one dimension, and making the correct matrix elements to be 1 or -4 such that we get the same set of equations as in the discretized Poisson Equation.

Here's what flattening a 2×2 square matrix looks like:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \xrightarrow{\text{flatten}} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix} \tag{2.10}$$

Thus, we get the matrix equation

$$MU_{flat} = -\rho_{flat} \tag{2.11}$$

Where M is the Matrix constructed for the Laplacian Operator $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$, and U_{flat}, ρ_{flat} are the flattened vectors of U, ρ
We get U_{flat} by solving this system of linear equations, and finally, get U by reshaping U_{flat} back into a square grid.
As an example, consider a 3x3 grid of points (so the flattened U and ρ vectors have 9 elements). The Poisson equation as a matrix equation, with $\rho = 1$ at the central point and 0 elsewhere will look like this

$$\frac{1}{-\Delta i \Delta j} \begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} U_{11} \\ U_{12} \\ U_{13} \\ U_{21} \\ U_{22} \\ U_{23} \\ U_{31} \\ U_{32} \\ U_{33} \end{bmatrix} = \begin{bmatrix} \rho_{11} \\ \rho_{12} \\ \rho_{13} \\ \rho_{21} \\ \rho_{22} \\ \rho_{23} \\ \rho_{31} \\ \rho_{32} \\ \rho_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.12}$$

Code Now for the coding part, define an nxn charge distribution grid with some appropriate value of $n \sim 50$. Set the following boundary charge distribution

```
cd_grid(:,:) = 0.0
cd_grid(n/2 + 10, n/2) = 1.0
cd_grid(n/2 - 10, n/2) = -1.0

cd_grid(1,:) = 1.0
cd_grid(n,:) = -1.0
```

You can also use any other charge distribution, but use this first to check if the code is working.
Write two subroutines, one to convert from grid to flat and other from flat to grid.

```
subroutine grid2flat(grid, flat)
    !convert the n x n grid to n**2 x 1 array
end subroutine

subroutine flat2grid(flat, grid)
    !convert the n**2 x 1 array to n x n grid
end subroutine
```

Carefully look at the discretized equation we derived for Poisson's equation. The left side of the equation is simply the discretized version of the Laplacian operator ∇^2. Below, we initialize the `operator` matrix with all zero elements. Complete the subroutine `make Laplacian operator` such that the matrix element `operator` is either 0.0, 1.0, or -4.0, based on the discretized Poisson equation.

```
subroutine make_laplacian_operator(operator)
    !input: operator, a N**2 x N**2 matrix with all it's elements zero

    !output: a N**2 x N**2 matrix with its element values such that
    !the U_flat vector, when matrix multiplied by the operator,
    !gives us the left hand sides of the set of equations
    !(one for each value of i, j) in the discrete Poisson equation
end subroutine
```

Now that we have converted our differential equation into a matrix equation, all that remains is to solve the matrix equation, which is

$$MU_{flat} = -\rho_{flat} \tag{2.13}$$

where M is your `operator` matrix, and ρ_{flat} is just `cd flat`. You need to obtain U_{flat} by solving this matrix equation, which is just a system of linear equations where the variables are the elements of U_{flat}.
Implement the function `solve system lineq` using either the Gauss-Seidel or Jacobi method. You can go back and decrease the value of `n` if your implementation is taking too much time to run.
Now for visualization. Convert the `U grid` to `U` matrix. Print this matrix in a text file like a 2D matrix.

```
open (unit=2, file = 'field.txt', status="unknown")
    do i = 1,n
        write(2,*) (U(i, j), j = 1, n)
    end do
close(unit=2)
```

Now plot this in gnuplot using the following commands

```
set pm3d map
set pm3d interpolate 2,2
splot 'field.txt' matrix
```

The code should take about a minute to run, depending on your system. If everything works great, you will get a plot similar to this:

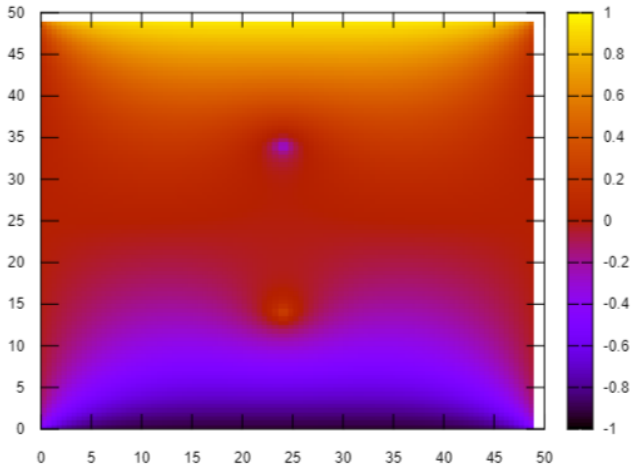

Figure 1: Looks great

Now you can try changing the color using different palettes. Also, try out different charge configurations.
I guess that sums up this week's task. Have fun!

Extra task 1 - faster processing In particular, a serious problem with the implementation above is that it wastes a LOT of memory; for $n = 50$, your operator matrix is of the size $10^4 x 10^4$ i.e 10^8 elements, most of which are zero.
To avoid this wasted space, we use sparse matrices. Sparse matrices cleverly store only the nonzero matrix elements to save space. Using this, you can go up to $n = 1000$ and more easily. You may use external modules or you can build your own. It won't be that hard, just store the location and value of non-zero elements of a matrix and you may need to redefine dot product of such matrices and of course the iteration methods.

```
integer :: rows(7*N**3 - 6*N**2), cols(7*N**3 - 6*N**2)
real :: data(7*N**3 - 6*N**2)
!The size of the columns will change according to our problem.

subroutine sparse_dot_vec(rows, cols, data, vec, dot)
    implicit none

    integer, intent(in) :: rows(7*N**3 - 6*N**2), cols(7*N**3 - 6*N**2)
    real:: data(7*N**3 - 6*N**2), vec(N**3 )
    real, intent(out) :: dot(N**3)
    integer :: index, row_index, col_index

    dot(:) = 0.0

    !A[x,y].b[y] = c[x]
    do index = 1, 7*N**3 - 6*N**2
        row_index = rows(index)
        col_index = cols(index)
        dot(row_index) = dot(row_index) + vec(col_index) * data(index)
    end do

end subroutine sparse_dot_vec
```

Try this and see how fast your code gets.

Extra Task 2 - Matlab As an extra task this week, you can familiarise yourself with MATLAB basics. MATLAB OnRamp is a very good beginning point to understand all the basic functionalities and syntaxes. Once you are comfortable with this try attempting *Problem 1* on MATLAB. If time permits, try *Problem 2* as well. Compare the speeds between your Fortran and MATLAB programs.