Miniproject - Algorithm, problems and suggestions

- 01 Algorithm

The problem for the mini project is solving a simple problem in electrostatics. Assume that you start with a distribution of heavy ions that generate an electric field that doesn't change in time (at least over the time that you are simulating). You can then work out the motion of a single electron in the field of those ions. The ions are represented as a charge density defined on a 2D grid and the electron is a particle that can freely anywhere within the grid.

To solve for the electric field due to the charge density you have to solve Poisson's equation

$$\nabla.\mathbf{E} = \frac{\rho}{\epsilon_0}$$

So long as your E field is "well-behaved" (this covers a multitude of sins but doesn't matter for this problem) you can use the Helmholtz decomposition to write E as

$$\mathbf{E} = \nabla\phi + \nabla \wedge \mathbf{A}$$

a combination of a irrotational (scalar potential) and divergence free (vector potential) part. For Poisson's equation you can immediately discard the divergence free part to gives two equations.

$$\nabla.\mathbf{E} = \nabla^2\phi = \frac{\rho}{\epsilon_0}$$

The easiest way of solving for the scalar potential ϕ is to use a finite difference representation of the derivative. You can derive this from a simple Taylor expansion of a 2D function f(x,y)

$$f(x+dx,y) = f(x,y) + dx\frac{\partial f}{\partial x}|_{x,y} + \frac{dx^2}{2}\frac{\partial^2 f}{\partial x^2}|_{x,y} + O(dx^3)$$

$$f(x-dx,y) = f(x,y) - dx\frac{\partial f}{\partial x}|_{x,y} + \frac{dx^2}{2}\frac{\partial^2 f}{\partial x^2}|_{x,y} + O(dx^3)$$

Expanding your function in x around the points x+dx and x-dx gives the two above series expansions, accumulating all terms in dx^3 or higher order in dx into the $O(dx^3)$ term. If you add those two functions together and rearrange for the second derivative in your function you get

$$\frac{\partial^2 f}{\partial x^2}|_{x,y} \approx \frac{f(x+dx,y) - 2f(x,y) + f(x-dx,y)}{dx^2} + O(dx^3)$$

If you then drop all of the higher order terms (the $O(dx^3)$ element) then you have an **approximation** to the second derivative in x for your function based on values on a grid with elements spaced by dx. You can perform an identical expansion in y to get the same expression in terms of y+dy and y-dy with the grid spacing dy. You can then write an equation for ϕ at grid point (i,j) in terms of the adjacent grid points.

$$\frac{\phi(i-1,j) - 2\phi(i,j) + \phi(i+1,j)}{dx^2} + \frac{\phi(i,j-1) - 2\phi(i,j) + \phi(i,j+1)}{dy^2} = \frac{\rho(i,j)}{\epsilon_0}$$

Each grid point is connected to the grid points surrounding it which are in turn connected to the grid points surrounding them etc. out to the edge of the grid. This means that you cannot rewrite the above equation algebraically to solve for ϕ(i,j). You have to pose the problem as a set of simultaneous equations, normally solved by forming a matrix and inverting it. The matrix formed from the above equation is a block tridiagonal matrix and there are many ways of solving such matrices. For simplicity we are going to set $\varepsilon_0 = 1$.

Here we are going to use the Gauss-Seidel method (https://en.wikipedia.org/wiki/Gauss–Seidel_method) which is an iterative method for inverting matrices. Iterative because you have to keep applying the same algorithm until you have converged close enough to the true solution. Since this isn't an algorithms course we're not going to ask you to derive the Gauss-Seidel update for this problem and the update expression for each point (i,j) is

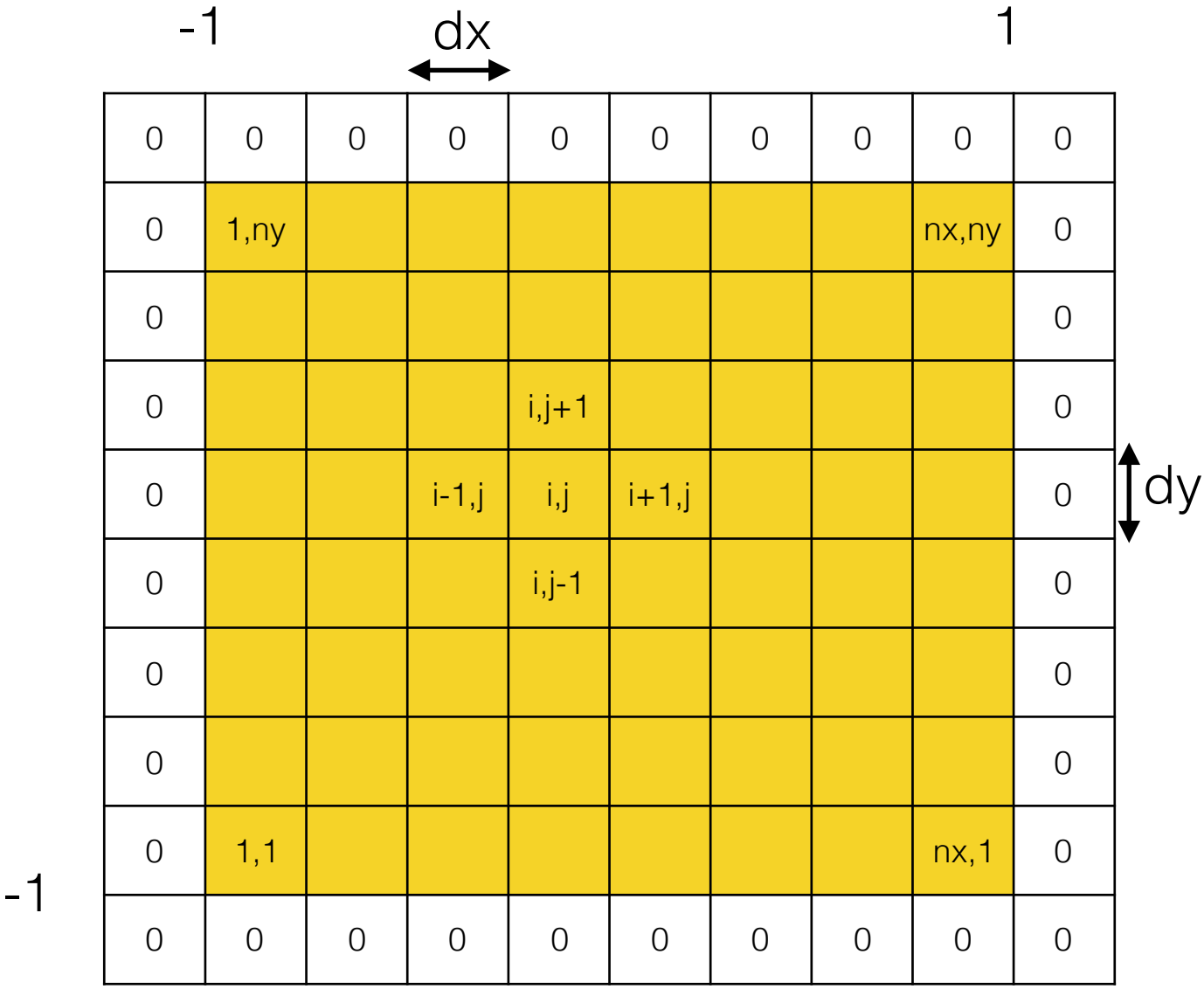
$$\phi(i,j) = -\frac{\rho(i,j) - \frac{\phi(i+1,j)+\phi(i-1,j)}{dx^2} - \frac{\phi(i,j+1)+\phi(i,j-1)}{dy^2}}{\frac{2}{dx^2} + \frac{2}{dy^2}}$$

It is very important to note that in Gauss-Seidel you always *immediately* update each point ϕ(i,j) and use that updated value when updating other values of ϕ. If you store the changes in ϕ(i,j) to a temporary value and then swap them all over when you have updated all ϕ(i,j) points that also works but is a substantially slower approach called Jacobi iteration.

Now that you know the algorithm, you want to know the grid that you will be solving it on. The grid isn't very complex. The domain consists of **nx** x **ny** elements and will have one strip of **ghost cells** or **guard cells** around the edge of the domain. These ghost cells contain the boundary values and are used when the update rule requires that you access an element outside the domain. So for example, updating ϕ(nx,ny) would require you to use elements ϕ(nx+1,ny) and ϕ(nx,ny+1). Without the ghost cells these would be outside your array and would cause a code error. While you could catch conditions like these that are invalid and do something else the use of ghost cells is a very simple way of allowing you to use exactly the same algorithm right up to the edge of the domain. Here, all ghost cells should just be set to 0.

We recommend that you use an array of size (0:nx+1, 0:ny+1) to hold ϕ.

While the domain has **nx** x **ny** elements, it is common for this type of code to represent a physical domain that is a known length that is not connected to the (numerical) number of grid points in the domain. In this case the domain runs from -1 to 1 in both X and Y directions. This means that **dx** and **dy** the size of the discretisation step used in the finite difference approximation depends on the number of cells used. It is important for symmetry that you define the axes correctly and you will be provided with code to make sure that this is done correctly and to provide you with **dx** and **dy**. The important element is that you want to make sure that -1 is at the very left hand edge of the domain and +1 is at the very right hand edge of the domain while you usually define ϕ (any other variables where relevant) at the centre of the cells. This means that the domain is actually one cell width longer than you might think. We would recommend using the provided code for domain lengths. The layout of the grid is shown in the below image.



Iterative methods like Gauss-Seidel require you to have some kind of measure for when you are “close-enough” to the real solution. First you use a measure of the error by simply calculating how far your solution is from the “true” solution

$$\frac{\phi(i-1,j)-2\phi(i,j)+\phi(i+1,j)}{dx^2}+\frac{\phi(i,j-1)-2\phi(i,j)+\phi(i,j+1)}{dy^2}-\rho=\text{error}$$

It is important to note that “true” here ignores the finite difference errors. This is the error purely due to the convergence of the Gauss-Seidel method. This error is not (itself) terribly helpful because how important it is depends on the size of the gradients in ϕ. What we want instead is a normalised error. Here we normalise it against the RMS value of the second derivative terms

$$e_{tot} = \Sigma \mathrm{ABS}(\frac{\phi(i-1,j) - 2\phi(i,j) + \phi(i+1,j)}{dx^2} + \frac{\phi(i,j-1) - 2\phi(i,j) + \phi(i,j+1)}{dy^2} - \rho)$$

$$d_{rms} = \sqrt{\Sigma(\frac{\phi(i-1,j) - 2\phi(i,j) + \phi(i+1,j)}{dx^2} + \frac{\phi(i,j-1) - 2\phi(i,j) + \phi(i,j+1)}{dy^2})^2}$$

Then keep iterating until the ratio $e_{tot}/d_{rms} < 10^{-5}$. Be careful about what can happen if d_{rms} is zero!

At this point we have a solution for the electric scalar potential ϕ. In order to calculate the force on a particle using the Lorentz force law you want the electric field.

$$\mathbf{E} = \nabla.\phi$$

This can also be obtained by a finite difference approximation. Returning to our previous expansions for f(x,y) and now subtracting them rather than adding them we obtain

$$E_x(i,j) = \frac{\phi(i+1,j) - \phi(i-1,j)}{2dx}$$

$$E_y(i,j) = \frac{\phi(i,j+1) - \phi(i,j-1)}{2dy}$$

These arrays only need to run from 1->nx and 1->ny because you no longer need ghost cells since you are not calculating derivatives of your E fields.

We now move on to how to move our particle. In a normal working code you wouldn't record the entire time history for the particle but here we want you to keep the whole history for the particle position, particle velocity and particle acceleration. To move our electron in the field we are going to use a velocity Verlet integrator **https://en.wikipedia.org/wiki/Verlet_integration#Velocity_Verlet**. There are good reasons to use this type of integrator but we're not going to go into them here. The update

sequence isn't very complex and for simplicity we again assume that mass and charge are magnitude 1.

As an initial condition you specify a particle position and velocity and given the particle's position you can calculate its acceleration from the Lorentz force law given the electric field at that point. Because the particle's location isn't locked to the grid in a working code you would normally want to interpolate the field from nearby grid points but this is difficult and error prone so we're going to simply use the electric field in the grid point that the particle is "in". This does mean that the particle experiences sudden changes in force as it crosses grid cells. This reduces the accuracy of the solution and can produce some slightly odd looking acceleration and velocity patterns but doesn't fundamentally change anything.

$$a^0 = q\mathbf{E}(\mathbf{x^0})$$

To calculate which cell your particle is in you can use

cell_x = FLOOR((part_x - 1.0_dp)/dx) + 1
cell_y = FLOOR((part_y - 1.0_dp)/dy) + 1

This works by working out how many dx or dy units your particle position is from -1 and then adding 1 (assuming that your real domain starts from array lower bound 1). The result is an INTEGER. You can then just find the field by using cell_x and cell_y to access your Ex and Ey arrays.

Once you have your initial acceleration you can start updating your particle positions, velocities and accelerations. The algorithm looks like

$$\mathbf{X}^{n+1} = \mathbf{x}^n + \mathbf{v}^n dt + \frac{1}{2}(\mathbf{a}^n)^2 dt^2$$

$$\mathbf{a}^{n+1} = q\mathbf{E}(\mathbf{x}^{n+1})$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + dt\frac{\mathbf{a}^{n+1} + \mathbf{a}^n}{2}$$

and you can see some of how it works. The update of position is similar to the familiar SUVAT equations, the update of acceleration simply relies on using the updated position. It is a bit stranger to see that updating the velocity uses the average of the current time acceleration and the next time acceleration but (if you are familiar with time integrators) the similarity to the midpoint method can be seen qualitatively (https://en.wikipedia.org/wiki/Midpoint_method).

Remember that we want to you keep the **whole** time history of the particle's position, velocity and acceleration so there's no problem with using both a^{n+1} and a^{n}.

Your code should take the particle and move it for 1000 iterations with dt set to 0.01. It is possible to work out a timestep based on the local velocity of the particle but it isn't terribly instructive for this problem. Just set it to 0.01. **Remember that mass and permittivity of free space are 1, and charge is -1.**

This then completes the idea of this code. It is related to a class of real codes called "test particle" codes that simulate the movement of particles in specified electric and magnetic fields and is also similar to the particle-in-cell (PIC) (https://en.wikipedia.org/wiki/Particle-in-cell) method where you update the electric (and usually magnetic) field in response to the movement of your particles (in some way).

- 02 Program details

Your code should

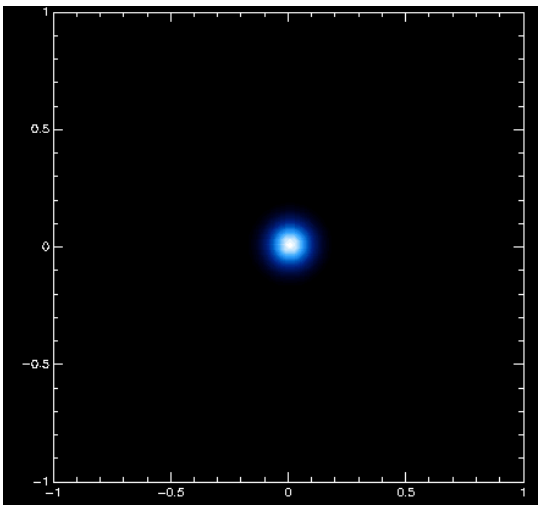
I. Take 3 command line options. **nx** and **ny** specify the number of grid cells in each direction as integers. Note that the domain should always run (-1,1) and (-1,1). It should also take **problem** which is one of the strings below specifying the initial conditions for the problem. If any of these parameters is missing your code should **STOP**

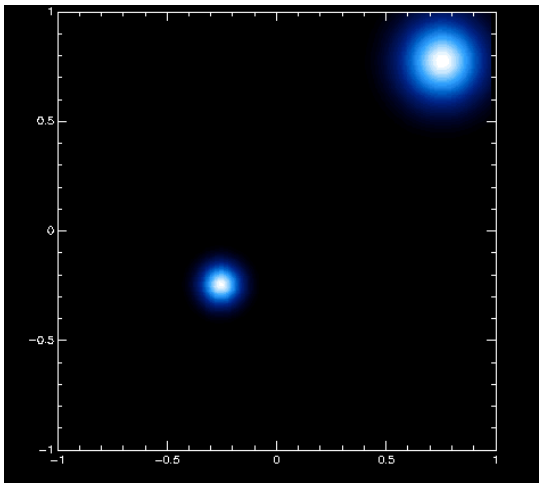
II. Set up initial conditions for the charge density field and the initial position and velocity of a single electron. This will be specified by a command line option as a string. The initial acceleration of the electron is completely determined by the electric field from the charge density field. You should write initial conditions that can be

   A. "null" - rho = 0 everywhere. Particle initial position is 0,0, particle initial velocity is 0.1,0.1

   B. "single" - rho = EXP(-(x/0.1)^2 - (y/0.1)^2), a single Gaussian peak at the origin. Particle position is 0.1, 0.0. Particle velocity is 0.0, 0.0

   C. "double" - rho = EXP(-((x+0.25)/0.1)^2 - ((y+0.25)/0.1)^2) + EXP(-((x-0.75)/0.2)^2 - ((y-0.75)/0.2)^2), two Gaussian peaks with different widths. Particle position is 0.0, 0.5. Particle velocity is 0.0, 0.0





III. Your code should use the algorithms in section 1 (note, these algorithms and no others !) to calculate the electric field from the charge density and then move the particle through 1000 steps with dt=0.01

IV. It should then output the following to a NetCDF file

   A. Charge density rho(1:nx, 1:ny)

   B. Calculated potential phi(1:nx, 1:ny)

   C. Ex(1:nx, 1:ny)

   D. Ey(1:nx, 1:ny)

   E. Particle position (0:1000) (x and y components, element 0 being initial condition)

   F. Particle velocity (0:1000) (x and y components, element 0 being initial condition)

   G. Particle acceleration(0:1000) (x and y components, element 0 being initial condition)

V. You should then provide a simple Python script to load the NetCDF file that you output and plot

   A. A pseudocolour plot of Ex (remember axes and colourbars!)

   B. A scatter plot of particle position in x vs particle position in y

VI. A simple shell script to

   A. Build your code

   B. Run your code with nx = 100, ny = 100 and problem “single”

   C. Run your Python visualisation script

- 03 Marking

1. What aren’t we marking?

   - Use of clever or advanced language features
   - Correctness of solution (within limits)

- A few minor mistakes that leads to you getting sensible answers that are different to ours isn't going to factor into marking
- Results that are obviously wrong are! Being able to code up an algorithm is a part of the skill set
- Performance is not marked, as long as your code runs in a sensible time (here definitely less than 10 mins).

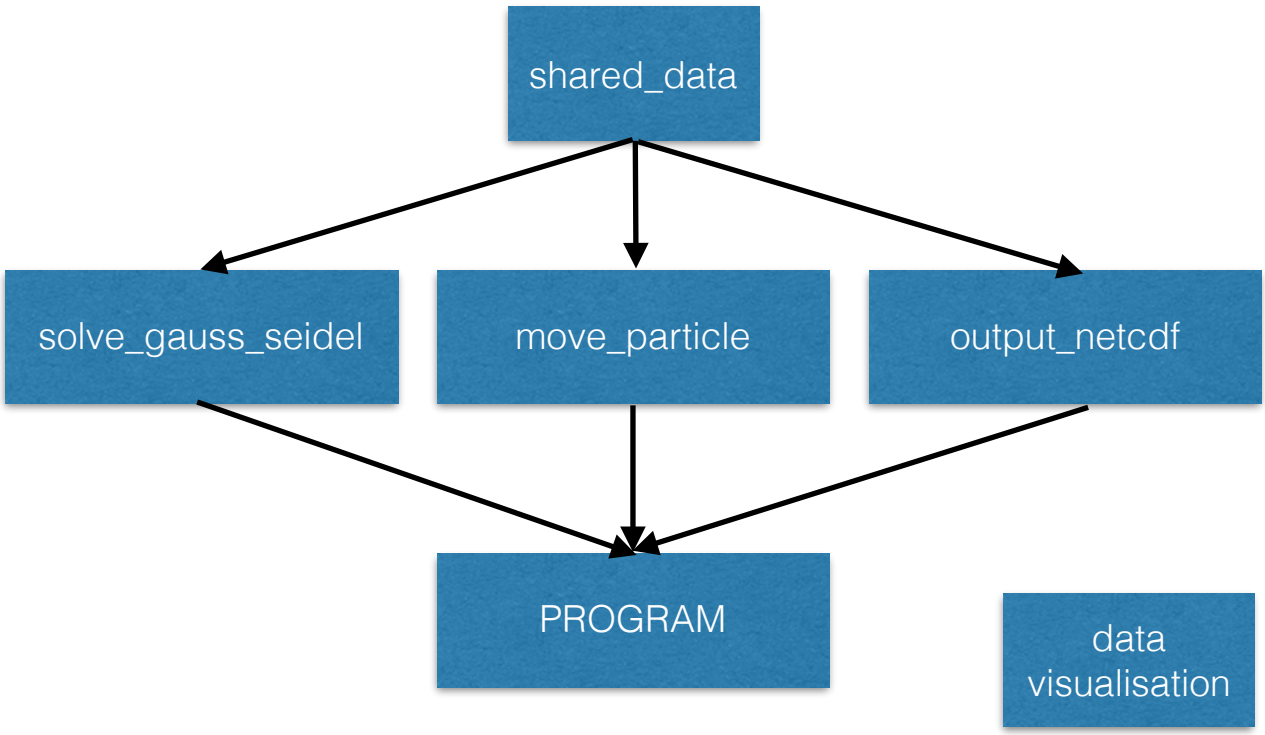
2. What **are** we marking?
    - Conformance to standards
    - Including memory leaks and array out of bounds errors
    - Comments and readability
    - Getting the right general answer
    - (Mostly) correct implementations of the algorithm
    - Evidence of working together

- 04 Advice - You don't have to use anything in this bit but it might be helpful

The work naturally splits up into 4 bits

- Gauss-Seidel solver
- Particle mover
- NetCDF output
- NetCDF input and visualisation

Suggest that you split up the work between the team members so that people each work on different bits. That will make it easier to put the work together and you might find this structure useful for the code itself. The relationship between the parts might look like



We recommend that you design how your code is going to work before you start writing any Fortran. You will want to design

1. How you are going to store data
    1. You will want arrays to store rho, phi, ex and ey. Since nx and ny will be run time variables you will want to use allocatable arrays for these.
    2. You can store them as module variables (variables defined in a module that other modules USE) or, if you want, you can put them in a derived type.
    3. Remember that if you passing them to functions or subroutines that the lower bound of an array will be remapped to 1 when you pass it to an array
    4. You will also want to store the 1001 elements of the particle position, velocity and acceleration somewhere. Since this information is so closely connected together you might want to use a derived type to hold it together
2. How you are going to structure your code

    1. How many files do you want?
    2. How many modules do you want? Remember that more than one module per file can be confusing unless the modules are quite closely related
    3. What functions are you going to write that people writing other bits will need access to? Are there any at all or are the bits self contained?
3. How are you going to work together?
    1. Github is quite convenient
    2. One person creates a github repository
    3. Invites the other people in the group as collaborators
    4. Try to work out how to minimise the amount of git merge conflicts that happen! (Hint git merge conflicts can **only** happen when two people change the same file!)
4. How are you going to check that you are getting the right answer ? (see marking section!)
    1. Potential will be peaked somewhere near the peaks in the charge density
    2. Remember that you can replace symmetric charge distributions with point charges at the centre point and you can approximately do this for this problem too
    3. You would expect an electron to be attracted to an ion so you would expect the electric field that you generate to produce the same sort of motion in general
    4. Note that your initial charge density has symmetry so you would expect your potential to have similar symmetry