1

Charged Particles Confined to a Sphere

Brandon Wilson

December 4, 1998

[1]Thanks to Dr. Matthew W. Choptuik for all his help (advice and code)

I. The Project and its Purpose

In this project, I have created a program that confines a user defined number of identical charges to a sphere of radius one. The forces that act on this system are the coulomb forces and drag forces on each particle. The coulomb force on each particle is, as expected, the sum of the coulomb forces from the other (n-1) particles. The drag force, proportional to the velocity of each particle, allows the system of n charges to reach an equilibrium. The user will be able to control the time steps (increments of time where the new positions are calculated), how close to actual equilibrium the system ends up, and the strength of the damping term (size of the damping constant). Controlling how close the system comes to equilibrium is very important, when using the program to find the equilibrium configuration of a large number of particles. The number of calculation perfromed, per iteration, is O(n^2). It is obvious that, for large n, the program will take an extremely large amount of time to execute. Reducing the accuracy of equilibrium, decreases the amount of time needed for the particles to reach a final position. The user can also decrease the amount of time needed, for the particles to reach equilibrium, by decreasing the time step value or increasing the damping constant.

One purpose of this project, is to allow me (and others) to analyze the equilibrium configurations of this interesting electrostatic problem. The most subtle purpose of this project is as a demonstration of how computer programming is a powerful tool for the physicist. This program is moderately difficult to code but compared to doing this problem by hand, for large n, it is trivial.

II. The Program

Before fully expaining the main program I would like to discuss several subroutines and a function that were borrowed or produced to make the source code more straight-forward.

- **subroutine force(r,f,n)**
  This subroutine calculates the total coulomb force for the n particles when it is given the x,y,z position arrays (3 x n array r) in the subroutine call. f is the 3 x n force array that is calculated and returned to the main program.

$$\mathbf{F_i} = \sum_{\mathbf{j=1}}^{\mathbf{n}} \frac{\mathbf{q^2}}{\mathbf{r_{ij}^2}} \tag{0.1}$$

- **subroutine dmmsmsa(a1,a2,a3,s1,s2,d1,d2)**
  This program is sent the matrices a1 and a2, as well as their dimensions (d1,d2). The matrix a3 is calculated and returned. It is of the form:

$$\underline{\underline{\mathbf{a3}}} = s1\underline{\underline{\mathbf{a1}}} + s2\underline{\underline{\mathbf{a2}}} \tag{0.2}$$

- **subroutine tangent(v,r,n)**
  Two column vectors (v and r)and their leading dimension are sent to this subroutine. It returns the tangent of the first vector with respect to the second, in the memory location of the first vector. This subroutine was vital since use of the non-tangential forces and velocities would have prevented equilibrium from occuring. To remove the radial part, I calculated the projection of v on r and subtracted that vector from v:

$$\mathbf{v} = \mathbf{v} - \frac{\mathbf{v} \cdot \mathbf{r}}{\mathbf{r} \cdot \mathbf{r}}\mathbf{r} \tag{0.3}$$

- **real*8 function ddotprod(v1,v2,n)**
  Function that returns the dot product of two column vectors sent to it. The only other value it requires in its call is the length of the column vectors. ddotprod returns:

$$\mathbf{v1} \cdot \mathbf{v2} \tag{0.4}$$

- **subroutine reflonsphere(v,n,r)**
  Tangential velocites and forces will not prevent newly calculated position vectors from being off the surface of the sphere. This subroutine is used to normalize the length of the position vectors to the radius (r) of the sphere, one in the case of this program. The positions of all particles are stored in a 3 x n array (v), one position vector per column).

- **subroutine gennpart(v,n)**
  This subroutine generates n length 3 vectors. The vectors are returned in a 3 x n array (v). This subroutine was used to create the initial positions of the particles.

- **subroutine dumpvec(quant,npart,label,unit)**
  Subroutine that dumps (outputs) an entire array, to where the user wants it (stdout, stderror,...) with a label and formatted. It was instrumental in the debugging of the source code. Its arguments are the array to be dumped (quant), the second dimension of the array (the number of particles), a character label, and a unit number that defines where the output is to be sent, in my case unit = 0 for standard error.

- **void sphplot(double *pos,int ncharge,double radius)**
  C subroutine that takes as its arguments a 3 x n position array (pos), the number of charges (ncharge) (the second dimension of the array), and the radius of the individual particles. These particles are drawn as spheres, with radius = radius, on a larger sphere. If the position array is sent for varying time steps, the result an animation of the particles as they come to equilibrium. This subroutine is only available on the SGI.

Notes: dumpvec and sphplot were written and supplied by Dr. Matthew W. Choptuik, who I would like to thank for all of his help in the development of this program. The code for all the subroutines and the main program are attached to this paper. They are also available in the /people/bwilson/project subdirectory. Most of the subroutines and the function call each other. For this reason, the object files corresponding to the above subroutines and function have been compiled into the projlib library. This library is located in the same directory given above.

The first step in creating the program was to find the equation of motion and the corresponding FDA. We begin with Newton's second law:

$$\mathbf{F} = \mathbf{ma} \tag{0.5}$$

acceleration is simply the second derivative of position and we know the forces, so Newton's second law becomes:

$$\ddot{\mathbf{r}}_i^n = \sum_{j=1}^{n} \mathbf{F_i} - \gamma \mathbf{v_i^n} \tag{0.6}$$

we are not particularly interested in the dynamics of the system. For this reason, we have arbitrarily assigned the charge to equal unity, with no loss of generality. All calculations were done with a choice of units such that, the mass of the particles is also unity. The summed force is simply the total coulomb force,

$$\sum_{j=1}^{n} \mathbf{F_i} = \sum_{\mathbf{j=1}}^{\mathbf{n}} \frac{\mathbf{1}}{|\mathbf{r_i} - \mathbf{r_j}|^{\mathbf{2}}} \tag{0.7}$$

where the sum over j gives the interaction between the ith particle and the other n-1 particles and i $\neq$ j (no self interaction) is implied. We now find the second order FDA of the acceleration:

$$\ddot{\mathbf{r}}_i^n = \mathbf{a_i^n} = \frac{\mathbf{v_i^{n+\frac{1}{2}}} - \mathbf{v_i^{n-\frac{1}{2}}}}{\mathbf{dt}} \tag{0.8}$$

dt is the mesh spacing in the time mesh for this FDA. The velocities are calculated at half mesh points because we are using the leap-frog method to calculate the velocities. The leap-frog method is known to be more accurate than the normal single mesh point FDA. The position vectors $\mathbf{r_i}$ are calculated at single mesh spacings. The leap-frog method is so named because we must leap-frog over positions to get new veloctitites. Now we must find the second order FDA of velocity:

$$\mathbf{v_i^{n+\frac{1}{2}}} = \frac{\mathbf{r_i^{n+1}} - \mathbf{r_i^n}}{\mathbf{dt}} \tag{0.9}$$

where the above statement is true because the n+$\frac{1}{2}$ velocity is equal to the difference of the position vectors about it divided by the time seperating them. Now we plug $\mathbf{a_i^n}$ (which is equal to the sum of the coulomb forces minus the damping force) into equation (0.4). The damping force contains a velocity term $\mathbf{v_i^n}$, what do we do with it? Simple, we plug in the equation for the average velocity using the half velocities:

$$\mathbf{v_i^n} = \frac{\mathbf{1}}{\mathbf{2}}(\mathbf{v_i^{n+\frac{1}{2}}} + \mathbf{v_i^{n-\frac{1}{2}}}) \tag{0.10}$$

with a little algebraic manipulation we arrive at a recursion relation for velocity. Algebraic manipulation of equation (0.5) reveals our recursion relation for position:

$$\mathbf{v_i^{n+\frac{1}{2}}} = \frac{\mathbf{dt} \sum_{\mathbf{j=1}}^{\mathbf{n}} \frac{\mathbf{1}}{|\mathbf{r_i}-\mathbf{r_j}|^{\mathbf{2}}} + (\mathbf{1} - \frac{\mathbf{dt}\gamma}{\mathbf{2}})\mathbf{v_i^{n-\frac{1}{2}}}}{(\mathbf{1} + \frac{\mathbf{dt}\gamma}{\mathbf{2}})} \tag{0.11}$$

$$\mathbf{r_i^{n+1}} = \mathbf{r_i^n} + \mathbf{dt}\mathbf{v_i^{n+\frac{1}{2}}} \tag{0.12}$$

It is with these two relationships that we can generate new positions at times l*dt (l = 0,...,nsteps). Where nsteps is the number of time steps (dt) contained in the interval [0,maximum time].

Usage: posphere $\langle$ n $\rangle$ [$\langle$ maxtime $\rangle$ $\langle$ nsteps $\rangle$ $\langle$ gamma $\rangle$ $\langle$ eps $\rangle$] Default values: 8 20 2.0d0 1.0d-16

Above are the command line arguments of posphere and their corresponding default values (n, the number of particles, must be entered). $\frac{maxtime}{nsteps}$ is the spacing of the time mesh and gamma is the damping constant. Together they make up the key parameter of this FDA (see equation 0.11). For the sake of brevity, I will now set them equal to a new variable (δ):

$$\delta \equiv \frac{\gamma dt}{2} \tag{0.13}$$

where,

$$dt = \frac{maxtime}{nsteps} \tag{0.14}$$

First, the program must parse the above command line arguments. The program then performs the following functions (in order):

1. Call gennpart to create the initial position vectors. These position vectors are then sent to reflonsphere to be normalized.
2. Calculate the value of δ and other constants used in the program (for more details see posphere.f).
3. Create initial velocity vectors and send them to tangent, to make them tangent to the sphere. In the case of this program, each component of the initial velocity vectors is arbitrarily set equal to one. Send initial velocity vectors to tangent, to remove their radial components.
4. Send position array to force subroutine, to create the initial force vectors. We must then send the position and the newly calculated force vectors to tagent, to make the force vectors tangent to the sphere.
5. initialize the chk2 vector. chk2 holds the most recent of the calculated position vector differences (r(t_{n-1}) - r(t_n)). chk1 holds the previous position vector difference, corresponding to the same particles. The difference of the two gives the current value of epsilon.

$$epsilon = |\mathbf{chk1} - \mathbf{chk2}| \tag{0.15}$$

6. Begin the main loop. The main, do while, loop runs until epsilon is less then the value given by the user on the command line or its default value. Once in the main loop, the first step is to initialize chk1 to chk2, before a new value of chk2 is calculated.
7. Call dmmsmsa to perform equation (0.11), thus calculating the new velocities. The next step is to call dmmsmsa to perfom equation (0.12). This creates the new positions from the newly calculated velocities.
8. Send the new velocity vectors to tangent.
9. Call dmmsmsa to find the latest value of chk2. Call dmmsmsa again to calculate the vector difference of chk1 and chk2. This difference is placed in chk1 which is then sent to ddotprod where it is dotted with itself. The square root of the value returned by ddotprod is the new epsilon.
10. With the new position vectors in hand, the force subroutine is called and the vectors returned are sent to tangent.
11. k is a counter variable corresponding to the current time step. The unix function dmod is then called. This function returns the real remainder of of k/100. If dmod(k,100.0d0) is zero the program outputs the time step (in multiples of 100) and the value of epsilon at that time. In this way, the user is aware of whether or not the particles are converging to their equilibrium positions. Iterate k for the next time step.
12. Before ending the loop, we have a call to sphplot to show the current positions of the particles. This allows us to see the movement of the particles towards their equilibrium positions. Immediatly outside the loop, another call to sphplot is found. In this one, the number of charges is given a negative value. To the sphplot subroutine this toggles the tumble mode. As the particles tumble, we can clearly see the equilibrium configurations.
13. The last step is output (to stderror). The user is shown the final positions of the particles, their final velocities, and the distances between the particles. The particle distances are calculated in a nested do loop. This nested loop is iterated in a way that prevents the distances between any two particles from being duplicated.

The discussion of a program containing an FDA would not be complete without a discussion of its key parameter. The accuracy of the approximation and whether or not the particles converge to an equilibrium is intimately related to the value of the key parameter. I found the significant values of δ to be:

$\delta < 0$ These values of δ violate the definition of a damping force. For ($\delta < 1$), the velocity remains in the computed direction but the forces are made negative (notice the ($1 + \delta$) in the denominator). These values of δ make the force repulsive. In such a case there will be no convergence. For (-1 < $\delta < 0$), the force is decreased slightly but the velocity is actually increased. Convergence is impossible in this case as well.

$\delta = 0$ This value of δ naturally eliminates damping in the system. This is the case of n free particles. There is no possibility of convergence.

$0 < \delta < 1$ These values of delta were found to be the most useful for convergence with excellent accuracy. There is a problem with this value of δ that will be discussed shortly.

$\delta > 1$ As delta is increased the accuracy of the equilibrium values decrease. The program must also perform many more iterations so that the appropriate value of epsilon is acheived. The reason is simple. As δ is increased, the velocities of the particles are decreased. The particles still move towards equilibrium, but do so very slowly. It causes the particles to under-shoot their equilibrium positions. Execution time increases because the computer requires more time to bring the particles to equilibrium. Since the computer cannot increase the user defined time spacings, it compensates by increasing the number of iterations.

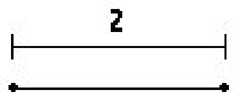
It was mentioned that there was a problem with the ($0 < \delta < 1$) values. The values of δ between zero and one are only feasible for smaller values of n (smaller forces and velocities). For large n, small values of delta increase execution time. This is due to increased net forces. **The user is warned that increasing deltas are required to keep execution time constant, as n is increased.** To make this program truly efficient it would be necessary to make δ a function of force. There would be a slight cost in the accuracy, but it would be well worth the decrease in execution time. The solution involves the incorporation of a viscous damping force. This would tend to counteract the increased force, caused by the smaller average distance between particles. With an understanding of posphere, we are now ready to discuss the configurations observed.

III. Discussion of Configurations

An arbitrary number of charges on the surface of a sphere will form a configuration that has the lowest possible total energy. Intuition leads one to believe that a great number of perfectly symmetrical configurations will result as n is increased. It is true that there are some perfectly symmetrical configurations but not many. We will start with the simple cases. All figures are 2-d views of the configurations.

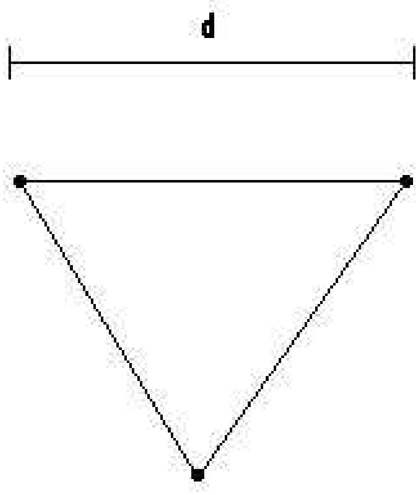
n = 1 This configuration needs no explanation. It is simply a free particle that will come to equilibrium because there are no forces acting on it.

n = 2 Intuition reveals the true equilibrium configuration. At equilibrium the particles are oriented at opposite points on the sphere. The distance between them is 2.0.



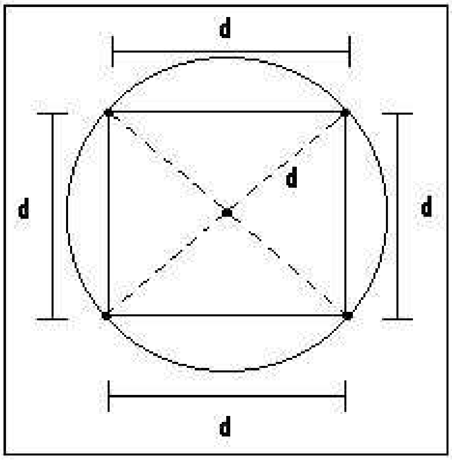
n = 2

n = 3 This configuration is also easy to guess. The three particles are the vertices of an equilateral triangle inscribed in a circle of radius one. The angle that the equalateral triangle makes with the equator is between -π and π. The distances between the particles is d = 1.7320508075688 ($\sqrt{3}$).



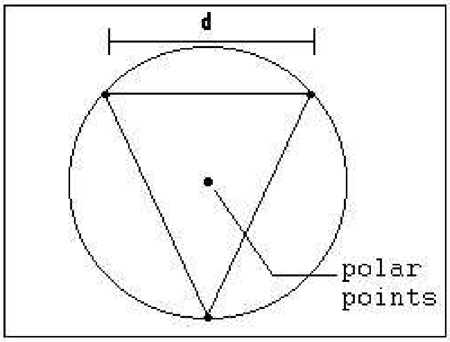
n = 3

n = 4 This is the last case of perfect symmetry (all particle distances equal). The particles make up the vertices of a tetrahedron at equilibrium. The distance between any two particles is d = 1.6329931618554. Note: the circle the triangle is inscribed in is **not** the equatorial triangle.



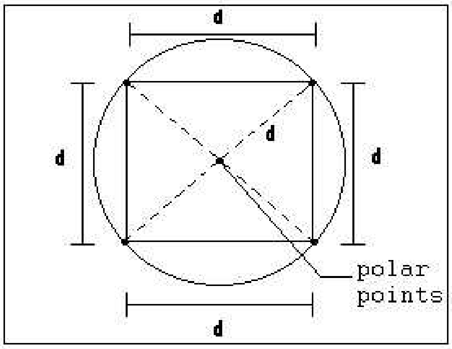
n = 4

n = 5 This configuration is not as easy to figure out as the previous but, it is still relatively simple. Two particles align themselves as poles on the sphere (the distance between them is 2). The remaining three form an equalateral triangle on the equator. These three assume the configuration of the (n = 3) case. The distance between them is, naturally, $\sqrt{3}$. The center point is the location of the polar particles. In other words, we are looking at a 2-d view from the top. A distance of 2 under this point is the south pole.



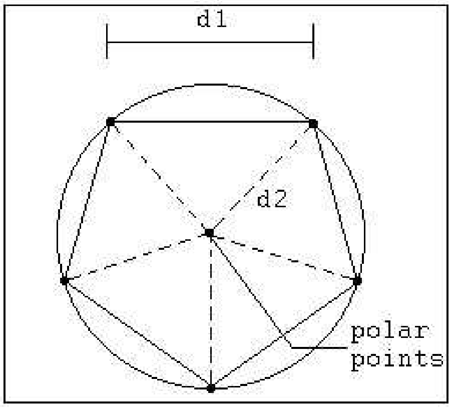
n = 5

n = 6 At equilibrium, the particles become the vertices of a perfect octohedron. Two particles are poles and the remaining four assume the shape of a square on the equator. It is obvious that 3 pair of particles will have a seperation distance of 2. All other particle distances are equivalent, being d = 1.4142135623730 ($\sqrt{2}$).



n = 6

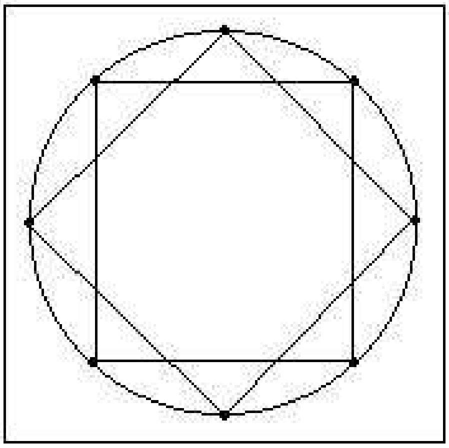
n = 7 We continue the trend of two polar particles and a symmetrical particle configuration on the equator. This time, the shape assumed by the particles on the equator is that of an equilateral pentagon. Besides the obvious distance of 2 between the polar particles, there are now three different particle distances. d = 1.9021130325903, d2 = 1.414213562373098, d1 = 1.175570504584945. The particle distance d is found between particles that are in the equatorial pentagon.



n = 7

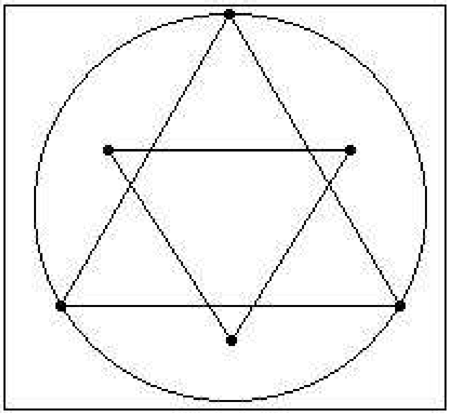
We now move on to some more interesting cases. I am omitting the distances between the particles. In the above cases, the values of the distances came from execution of posphere. If one is interested in finding the particle seperations, all they must do is run posphere with the appropriate values.

n = 8 This is a very interesting case indeed. Intuition would tend to make one believe that the equilibrium configuration is a perfect cube. In many cases, symmetry and equilibrium (lower energy) go hand in hand. Here intuition fails. The configuration is not a perfect cube; it is a cube with one face turned 45 degrees with respect to the other (see figure below). Each square lies in a seperate plane and the planes are parallel.



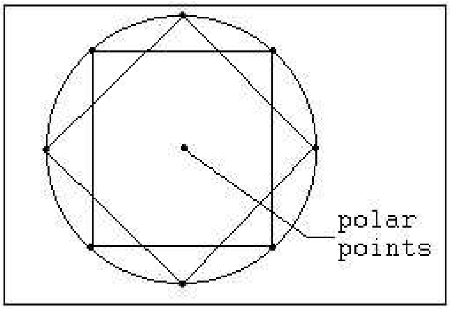
n = 8

n = 9 This is also an interesting case. The configuration is of 3 equilateral triangles on different planes. The vertices of the triangles correspond to charges on the sphere. The largest triangle lies in the equatorial plane and the other two are an equal distance above and below the equatorial triangle. The interesting part of this case is purely geometrical. It is difficult to imagine that the natural equilibrium configuration of any system could produce such beautiful geometry.



n = 9

n = 10 This case is simply the reproduction of the (n = 8) case with two simple modifications. The axis through the center point of the squares intersects the two added charges. We have the (n = 8) case with a north and south pole.



n = 10

1. As the number of charges increases past 10 the configurations become less appearent to the human eye. For this reason, I will conclude this discussion.

First, I will discuss the very interesting (n = 60) case.

n = 60 This produces the famous bucky ball (C_{60}) configuration. It's stability as the third major form of pure carbon helps to show that posphere works properly. If posphere can reproduce C_{60}, it must be producing proper equilibrium configurations.

In conclusion, reproduction of the C_{60} configuration using posphere demonstrates something that is seen time and again (especially by physicists). Equilibrium (minimum energy) configurations are reproduced, identically, time after time in a myriad of systems. This fits with a truth told to scientists. Any system (excluding some quantum systems) will exist in the configuration that produces the lowest total energy. We have seen that posphere works according to specs. and produces some interesting and geometrically beautiful configurations. It is also appearent that the computer is a boon to the physicist. Using (n = 60) as an example, the computer performed a task, that would take a scientist days, in a matter of minutes. Not only that, the computer doesn't have to start from scratch when the configuration of another number of particles is needed.

Sources

1. http://oaks1.oakwood.edu/buckyball.html
Information found here for the bucky ball.

2. http://www.seanet.com/ ksbrown/kmath005.htm
Information found here was invaluable for testing purposes and for the discussion of configurations.