Computational exercise: The motion of a charged particle in an electric field
PH 220

Objective: Write a computer code that reads in the locations and charges of several point charges, as well as the charge, mass, initial position, and initial velocity of an additional charged particle. Then determine the trajectory of the particle through the fields using Euler's method, for a specified time.

Background: We've now done several exercises where we have found the electric field from both discrete and continuous charge distributions. For this exercise, we are now going to consider the motion of a charged particle in a complicated electric field.

The fundamental physics behind this problem is straightforward: the force acting on the charged particle is given by

$$\vec{F} = q\vec{E}$$

and once we know the force acting on the particle, we can find the acceleration

$$\vec{a} = \vec{F}/m$$

Knowing the acceleration, we can get the velocity and position through time by integration. Here, though, is where things get complicated. If the electric field is relatively simple, those integrations can be performed explicitly. However, if the electric field is very complicated, such as the field from a large assembly of point charges, the integrations become impossible. This is where numerical methods step in.

In this assignment, we are going to step our particle forward in time using Euler's method. This is a rather simple method, although it does come with some risks, which I will describe a little later. The basic principle is easy enough. For a very small time interval, the acceleration and velocity of an object will be approximately constant. This being the case, we can just use the constant acceleration equations to take a very small step forward in time. After taking that step, we reevaluate the acceleration (based on our new position), and repeat the process. In other words, for a very small time interval dt, we simply do something like the following:

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

These equations will give us the approximate position and velocity of the particle at the end of the interval. We then use the new position to re-evaluate the electric field, force, and acceleration.

Now for the risks. In the limit where the time step dt becomes infinitesimally small, Euler's method will provide an exact solution. In practice, though, very very small time steps mean a lot of computation time. So what do we do? We compromise. We decide that a small amount of error is "acceptable" in exchange for a reasonable run time on the program.

One last bit of advice before you get started: since you will be evaluating the electric field at a very large number of points, I highly recommend that you write a function or subroutine that returns the field. This is not an absolute necessity, although it may help your code become more readable in the future.

I have no preference what method you use to find the electric field (superposition, as in assignment #1, or via the gradient as in assignment #3).

Exercise requirements: Write a computer code that does the following:

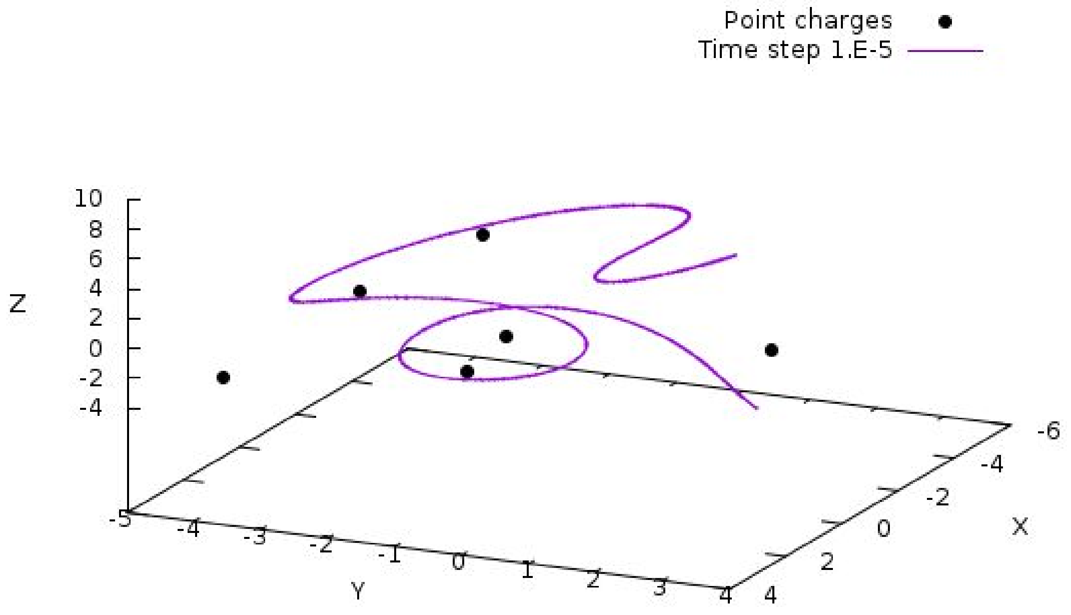
1. Reads a file specifying the number of source charges, the charge (in nC) and the coordinates (in cm) of each of the source charges, the charge (in nC) and mass (in grams) of the particle that we want to track, its initial coordinates (in cm), its initial velocity (in cm/s), and finally, the duration and time step to use in the simulation (in seconds). This input file will be formatted as follows. The first line of the input will consist of a single integer specifying how many source charges there are. The next lines, one for each source charge, gives the charge in nC, the x-coordinate in cm, the y-coordinate in cm, and the z-coordinate in cm. The source charge lines are followed by a line specifying the mass and charge of the particle we want to track, in grams and nC respectively. The next line specifies the initial x-, y-, and z-coordinates of this particle, in cm, and is immediately followed by a line that specifies the x-, y-, and z-components of the particle's initial velocity, in cm/s. The final line gives the duration of the simulation, in seconds, and the time step to use, also in seconds. Here is an example input file.

```
6
  -3.000   1.323  -0.458   0.498
   2.000  -1.685   2.254  -2.211
   1.000   3.844   0.003   1.112
  -5.000  -4.451  -3.239  -0.731
   2.000   3.332  -3.854  -2.074
   1.000   2.133  -2.298   3.287
1.15E-1 3.323
   3.000   4.000      0.000
   1.000   0.000      2.000
   4.000   1.E-5
```

2. Determines the trajectory of the particle using Euler's method, as explained above.
3. For each time step, writes out the current time in seconds, the coordinates of the particle in cm, and the velocity components of the particle in cm/s, formatted as follows:

```
0.000000   3.000   4.000   0.000   1.000  -0.000   2.000
0.000010   3.000   4.000   0.000   1.000  -0.000   2.000
0.000020   3.000   4.000   0.000   1.000  -0.000   2.000
0.000030   3.000   4.000   0.000   1.000  -0.000   2.000
0.000040   3.000   4.000   0.000   1.000  -0.000   2.000
0.000050   3.000   4.000   0.000   1.000  -0.001   2.000
0.000060   3.000   4.000   0.000   0.999  -0.001   2.000
0.000070   3.000   4.000   0.000   0.999  -0.001   2.000
0.000080   3.000   4.000   0.000   0.999  -0.001   2.001
0.000090   3.000   4.000   0.000   0.999  -0.001   2.001
0.000100   3.000   4.000   0.000   0.999  -0.001   2.001
```

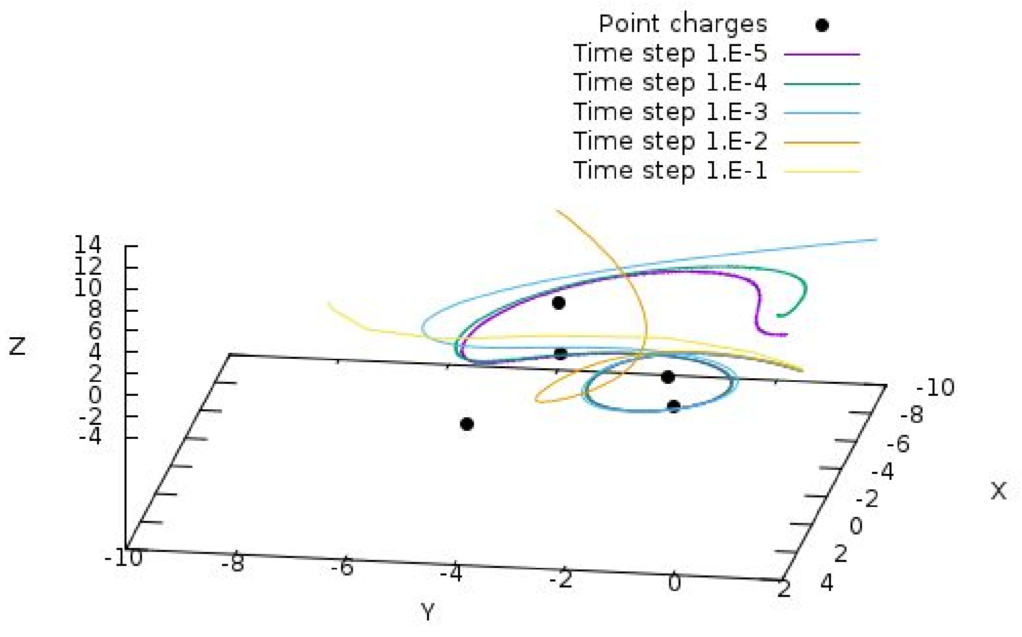
Note that since we are simulating the particle's trajectory for 4 second, but with a 0.00001 second time step, your output file will have 400,000 lines. Incidentally, the trajectory for this particle among these source charges looks like the following:



Validation: You can verify whether your code is working correctly or not by feeding it the example input file above, and seeing whether it reproduces the following results at the appropriate times (note that these lines are taken from the output for the given input file - I did not use a 0.5 second time step!):

```
0.000000   3.000   4.000   0.000   1.000   0.000   2.000
0.500000   2.042   2.732   1.503  -6.327  -6.710   3.310
1.000000   2.265  -0.246  -1.288  -4.665  14.885   6.929
1.500000  -1.095  -4.543  -1.347 -18.515  -3.039  -8.430
2.000000  -2.914   0.475   4.154  10.991   3.722   8.697
2.500000   1.381   1.171   6.485   5.777   0.693   2.237
3.000000   3.134   1.694   7.374   1.620   1.407   1.651
3.500000   3.238   2.499   8.206  -1.078   1.719   1.634
```

Now is probably a good time to demonstrate what happens if the time step is too large: the approximation gets poor, and the resulting trajectory is not correct.



In this plot (viewed from a slightly different angle than the last), the various colored lines represent Euler solutions using progressively larger time steps. You will note that each line starts at the same place (lower right side). The purple line represents the smallest time step (incidentally, I also ran this with a time step of 10^{-6} seconds, and there was very little deviation between that run and the one with a time step of 10^{-5} seconds). The green line represents a time step of 10^{-4} seconds. You can see that it tracks quite well with the purple line for the most part, but starts to deviate toward the end. The blue line represents a time step of 10^{-3} seconds, and does well through the first loop, but shortly thereafter doesn't do quite so well. The orange line

(10^{-2} seconds) doesn't even make it through the first loop before spiraling out of control. The yellow line (10^{-1} seconds) is right out. Eventually, every Euler simulation will become unstable and fail. It just depends on how long the simulation lasts, and the relative size of the time step.

Grading rubric: Your grade on this exercise, out of 50 points total, will be calculated based on the following criteria:

- Your code accurately tracks the projectile for a test input file that I provide. I will check the position of the particle at 5 different times, with two points for each coordinate. (30 points) Note that since everyone is using the same time step, everyone should be getting the same results!
- Your code is well organized and "pretty". (3 points)
- The output is formatted as required. (2 points)
- You have included all of the following comments in your code:
  - A header that describes what the code does and how to use it. (5 points)
  - A description of each of the variables in your code. (5 points)
  - Frequent descriptions of what your code is doing, including any numerical methods that you are employing. (5 points)