Atomic Simulation

A program was written to simulate, in two dimensions, the interaction of idealized atoms. The atoms experience and exert only Van der Waals forces. These forces are estimated using the Lennard-Jones potential:

$$V(r) = 4\epsilon \left((\sigma/r)^{12} - (\sigma/r)^{6} \right) \qquad [1]$$

where r is the distance between two atoms, and ϵ, σ are chosen heuristically so as to be in agreement with experimental evidence. To find the force in terms of the distance between two atoms we take the differential with respect to r.:

$$\begin{aligned} F(r) &= d/dr\left(4\epsilon \left((\sigma/r)^{12} - (\sigma/r)^{6} \right) \right) \\ &= 4\epsilon \sigma^{6} \left(d/dr\left(\sigma^{6} r^{-12} - r^{-6} \right) \right) \\ &= 4\epsilon \sigma^{6} \left(-12 \sigma^{6} r^{-13} + 6 r^{-7} \right) \\ &= 24\epsilon \sigma^{6} r^{-7} \left(1 - 2 \sigma^{6} r^{-6} \right) \end{aligned}$$

To find the effect one of the atoms involved, we find the change in velocity along the line between the two atoms to be the force divided by the mass.:

$$dr/dt = F/m$$

This calculation produces the vector which drives the update() function of the simulation.

The simulation begins by populating a "cell" with the simulated atoms. The cell is a rectangular(usually square) two-dimensional space in which each atom is represented by a uniquely colored circle. When displayed, the representation of the cell can be "reflected" into what appear to be adjoining areas of space occupied by identical atoms. Each atom in the cell feels not only the effects of the presence of other atoms in the cell, but those in "reflection" cells as well, whether or not they are displayed. The cell repeats itself, as if it represented a nano-scale doughnut-shaped universe. The first atom to be place in the cell can be placed anywhere, but subsequent atoms must not be placed on top of existing atoms. A new atom will only be added to the cell if it can be placed at least an atom's width away from any other atom. Otherwise, such a placement could result in a potential energy too high for the simulation to handle. It would be akin to a high energy collision.

Besides infinity, there is another "zero" point for the distance between any two atoms. It is the point when the above force equation reaches zero.:

$$F(r) = 0 \quad = 24\epsilon \sigma^{6} r^{-7} \left(1 - 2 \sigma^{6} r^{-6} \right)$$

$$24\epsilon \sigma^{6} r^{-7} \left(2 \sigma^{6} r^{-6} \right) = 24\epsilon \sigma^{6} r^{-7}$$

$2\,\sigma^6\,r^{-6} = 1$

$r^6 = 1/(2\,\sigma^6)$

$r = 1/(2^{1/6}\,\sigma) \doteq .89/\sigma$

To avoid unnecessary complications when setting up a scenario, we will only add atoms that are at least $.89/\sigma$ away from any other atom. This distance will also be used as the radius of each atom.

To improve computation time, an approximation will be used when computing the force that two atoms place on each other. Any two atoms that are at least 2.5σ apart from each other will be regarded as infinitely far, i.e.: they will exert zero force on each other. At this distance the force is approximately $1/60^{th}$ its maximum value [1]. The value is exactly:

$$F(2.5\sigma) = 24\in \sigma^6\, r^{-7}\,(1 - 2\,\sigma^6\, r^{-6})$$

$$= 24\in \sigma^6\,(2.5\sigma)^{-7}\,(1 - 2\,\sigma^6\,(2.5\sigma)^{-6})$$

$$= 60 \in (1 - .008192)/\sigma$$

$$\doteq 60 \in / \sigma$$

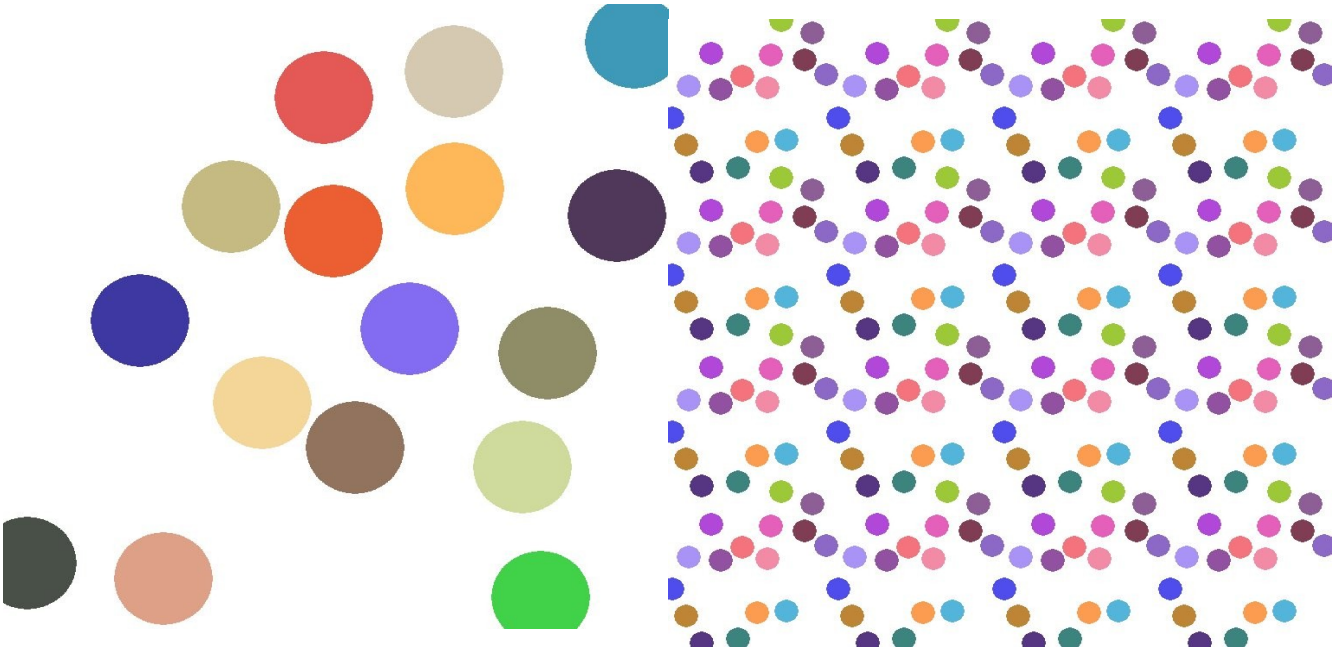
Any two atoms at a distance of at least 2.5σ will exert no force on each other in the simulation. In order to maintain continuity in F(r), an offset equal to $60\in/\sigma$ will be subtracted from the force applied by any two atoms closer than 2.5σ.

8.2 Approach to Equilibrium

8.2.a A simulation was begin with 16 atoms and Lx = 6. The timestep was set to .0001 because it was found that higher timesteps allowed atoms to encroach upon one another in such a way as to cause huge variation in the potential energy, often causing spikes greater than the computer's maxfloat. Below is a screenshot of the simulation at initialization. This screenshot shows only the main cell. Some atoms are partially off the edge. The partially obscured part would theoretically be protruding from the other side of the cell (wrapping around), but is not shown because of complications in the graphing code. When reflections are shown this problem is not an important factor because all atomic motion between cells appears fluid and continuous.

The figure below left shows the 16 atoms randomly distributed at the beginning of a simulation. Each atom is uniquely colored to make the simulation easier to follow visually. The trajectories of the atoms are relatively fluid. They become more fluid if fewer atoms are simulated.

The simulation is able to display the cell alongside a somewhat arbitrary number of reflections. Each atom feels the effect of each other atom +3 reflections (whether displayed or not). Typically, only one of those four is close enough to exert any appreciable force. Besides those 4, each other reflection is merely a display feature. Simulations have been done with up to 40X40 reflections. The actual simulation is not altered in any way by the presence of these reflections. Another 16-atom simulation is shown below (right) with 15 reflections (all sets of reflections are configured in squares -- 16 total cells, or 9, 4, 1, 25, etc.).

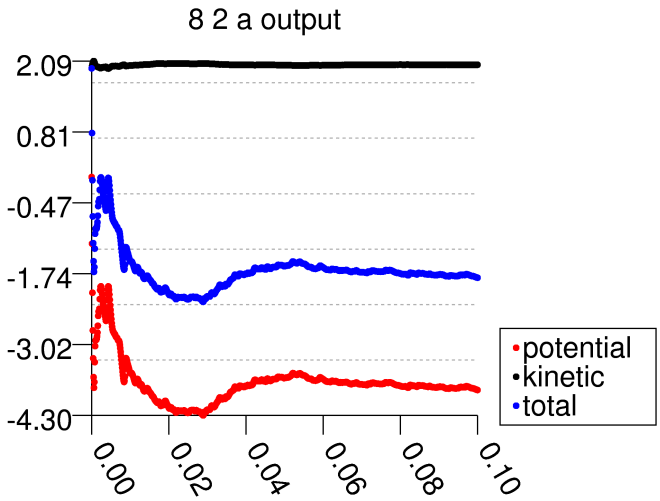


The total energy was computed at each time interval and was observed to be approximately conserved. The following is the potential and kinetic energy of the system during one simulation. On the left is time (running down). Next is the average potential energy so far (using the LJ-potential). Next is the average kinetic energy so far (using cv^2, where c is some coefficient to put the kinetic and potential energies in the same order of magnitude). Next is the sum of the two (the total energy), and finally is the number of timesteps completed so far.

time	potential	kinetic	total	steps
9.999900	-1.526787	0.283056	-1.243731	2
9.999800	-2.046015	0.285324	-1.760691	3
9.999700	-2.473798	0.287664	-2.186134	4
9.999600	-2.888921	0.289965	-2.598956	5
9.999500	-3.211174	0.292131	-2.919044	6
9.999400	-3.312547	0.294079	-3.018468	7
9.999300	-3.043300	0.295739	-2.747561	8
9.999200	-2.455369	0.297057	-2.158312	9
9.999100	-1.997987	0.298008	-1.699979	10
...				
8.689500	-1.854071	3.879727	2.025656	13106
8.689400	-1.847987	3.892473	2.044487	13107
8.689300	-1.847731	3.905226	2.057495	13108
8.689200	-1.847317	3.917972	2.070654	13109
8.689100	-1.845445	3.930728	2.085282	13110
8.689000	-1.844205	3.942972	2.098767	13111
8.688900	-1.834018	3.948967	2.114949	13112
8.688800	-1.832934	3.954949	2.122015	13113
8.688700	-1.832707	3.960928	2.128222	13114
8.688600	-1.832683	3.966906	2.134223	13115
8.688500	-1.832591	3.972883	2.140292	13116

As shown, each of the energies and the total energy is not perfectly constant, but it stays relatively constant -- at least enough for the simulation to look natural.

Below is a graph of each type of energy as a function of time:



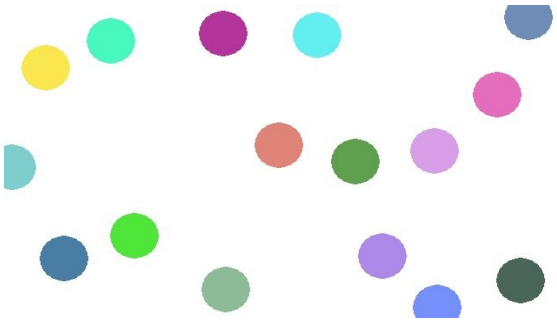
The above shows the average potential, kinetic, and total energy as a function of time (in seconds), for a cell of size 6X6, containing 16 atoms with originally random positions and

velocities, where each datapoint comes from one timestep, and each timestep is .0001 seconds. The term "seconds," is used loosely here as the simulation happens in a unitless environment.

A relative equilibrium is reached by about .05 seconds, however, it is striking that the kinetic energy is so constant relative to the potential energy. The "total" energy is merely the sum of the other two.

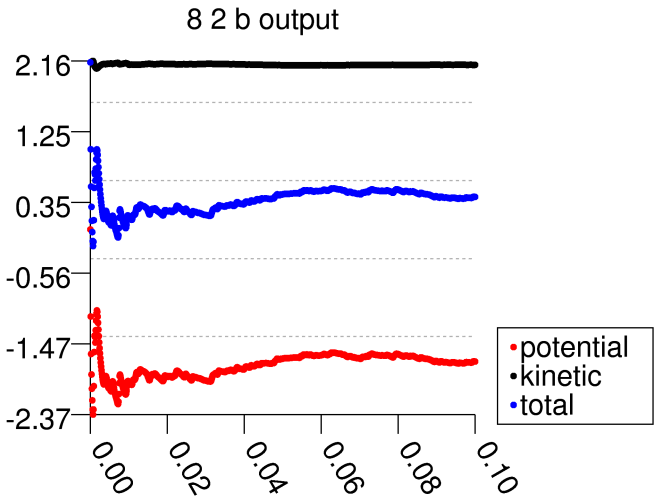
8.2.b

This simulation differs from 8.2.a only in the size of the cell, and the fact that a new set of 16 randomly distributed atoms was generated. The cell in this run is 12X6.:



The question is, does the system go toward equilibrium or chaos?, and how does it compare to the previous example?

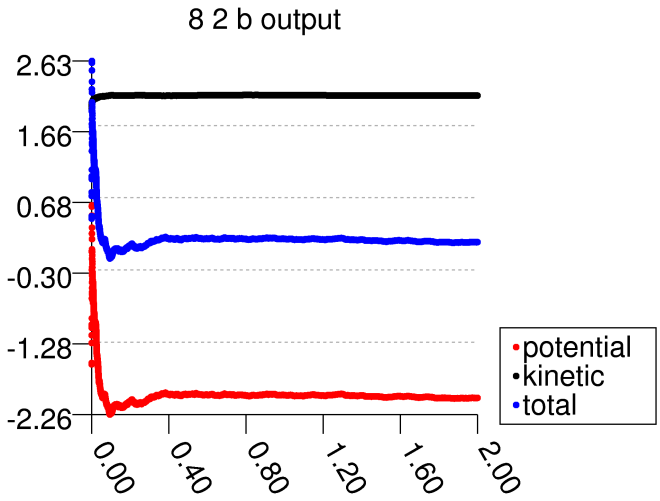
The graph below shows the average potential, kinetic, and total energy over the first .1 seconds.:



The graph above shows the energies of 16 atoms in a cell of size 12X6, with originally

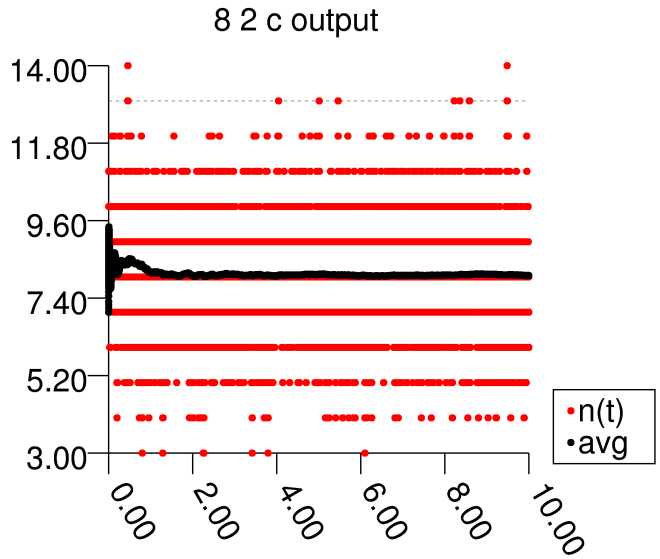
random positions and velocities. A relative equilibrium appears to be reached after approx. . 04 seconds.

Now let us examine the behavior of such a system over a longer timeperiod, such as 2 seconds.:



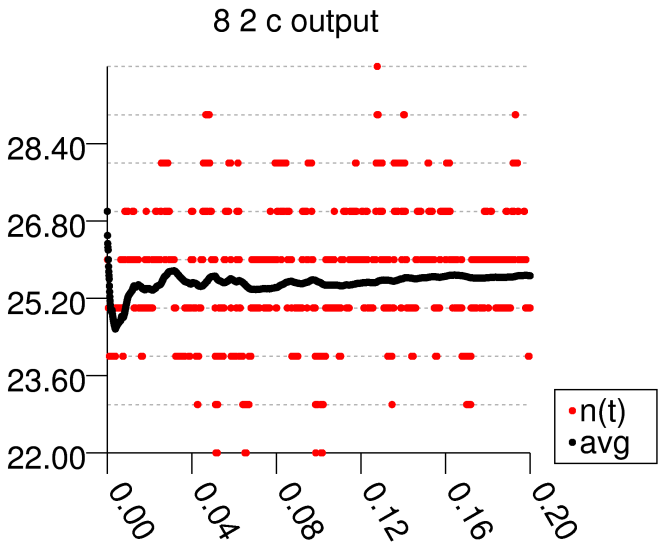
As can be seen in the graph above, the system definitely appears to find an equilibrium. At least during the simulated 2 seconds the system does not exhibit chaotic behavior.

8.2.c. Now let us examine the number of atoms in the left half of the 12X6 sized cell. Let n(t) be the number of atoms in the left half at time t.



The above graph shows that the system never approaches an equilibrium. The average is consistently 8, while the values of n(t) continue to vary mostly between 6 and 10. The deviation from 8 does not appear to change over time.

Next the same experiment was attempted with 64 atoms, but it was not possible to randomly place 64 atoms in the cell without having collisions in the initial conditions. 54 atoms were successfully place within the cell. Below we see the results of n(t).



A shorter time period was used because the high number of atoms makes each timestep expensive. As can be seen above, the larger system shows the same pattern of consistent deviation from the mean (25.5, if one allows a non-integer mean). The difference is that the deviation is relatively less than in the previous run. n(t) appears to stay mostly within the range 24 to 27, a range of 3 compared with a mean of 25.5 is less significant than a range of 4 compared to a mean of 8.

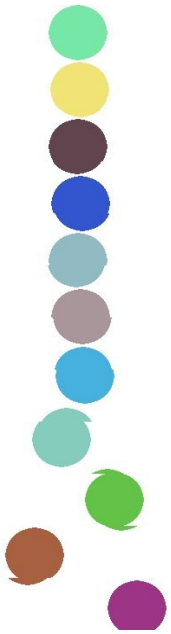
8.3 Sensitivity to initial conditions

8.3.a. A simulation was run with these initial conditions:

x	y	x'	y'
5	5/11	10	0
5	15/11	10	0
5	25/11	10	0
5	35/11	10	0
5	45/11	10	0
5	55/11	10	0
5	65/11	10	0
5	75/11	10	0
5	85/11	10	0
5	95/11	10	0
5	105/11	10	0

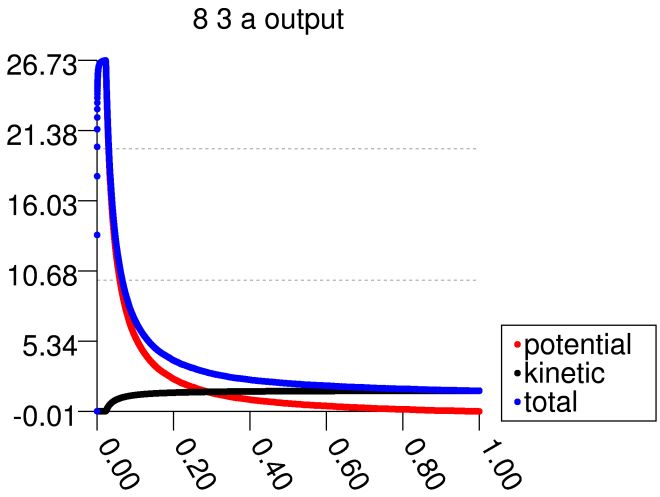
A velocity of 10 was used (as opposed to the suggested velocity in the text) because of the much smaller timestep being used (.0001).

The simulation appears to begin in a state of equilibrium with all atoms moving in parallel at the same speed. However, almost immediately, tiny perturbations appear in the y-axis motion of several atoms. After approximately .023 seconds, these perturbations evolve into a sudden rupture of the clean line of atoms.:



The rupture sends the atoms in all directions, very quickly erasing any order that was previously there. The screenshot above was snapped just as the rupture began. The non-spherical shape of some atoms is an irrelevant artifact of the screenshot capturing process.

Does the system come to an equilibrium at some point after the rupture?



It is easy to see from the above graph that the system does indeed come to an equilibrium. The initial conditions were extremely lopsided in favor of potential energy over kinetic. For that reason it was not stable. In a real system the same thing would be true --order devolves into chaos. Such a configuration is not stable. The initial conditions in the simulation were not stable for two reasons: 1. lopsided toward potential, and 2. roundoff errors. If there had never been any roundoff errors, then it is likely that each atom would never alter its trajectory, resulting in a "stable" system. To be truly stable, the state of a set of atoms should return after any minor perturbation. A stable configuration can also be considered a global minimum, as opposed to a local minimum.

8.3.b. A simulation was run with these initial conditions:

x	y	x'	y'
5	5/11	10	0
5	15/11	10	0
5	25/11	10	0
5	35/11	10	0
5	45/11	10	0
5	55/11	9.9	.1

```
5  65/11    10    0
5  75/11    10    0
5  85/11    10    0
5  95/11    10    0
5  105/11   10    0
```

The atoms begin with what appears to be the same configuration as in 8.3.a. The difference is that the locally stable initial conditions decay more rapidly into entropy. The perturbations begin almost immediately, as before. The rupture occurs, however, after a mere .0045 seconds.

If 8.3.a represented an unstable configuration, this run represents a less stable one. It was inevitable that both configurations would devolve into entropy.

8.3.c The simulation from 8.3.a was run until .023 seconds and then reversed to see if the atoms would return to their original configuration. They did indeed return to their previous configuration, but only in passing. The moved into a line and didn't stop. The set of atoms never did return to all sharing a velocity vector.

The simulation was run again, this time stopping at .022 seconds. The reversal again put them back in the line configuration. The atoms appeared to all have the same velocity vector for two or three timesteps.

This simulation was run again with a reversal at .021 seconds. The results were the same as for .022s. Reversing at .02 seconds made the atoms line up for a few more timesteps. Finally, the simulation was reversed after only .01 seconds. The atoms remained lined up going in the reverse direction for approximately .014 seconds before the line ruptured.

One more test was run using a smaller timestep of .00001. In this scenario the line of atoms ruptured after .0194 seconds. The simulation was reversed at this point. The atoms did return to the line, but only held the configuration for about .00055 seconds.

8.3.d. These conditions were covered thoroughly in 8.3.c.

8.3.e. It is highly improbable that the computed trajectories are the same as the would-be true trajectories. These atoms are all idealized, and our calculations do not incorporate any quantum mechanics. What can be said, however, is that true trajectories of atoms in nature "look like" the simulated ones. This fact was referred to earlier when it was mentioned that atoms in nature, like the simulated ones, will not remain long in a configuration of local stability, i.e.: a state which is highly sensitive to perturbations. A state that is stable in nature should be stable in the simulation as well.

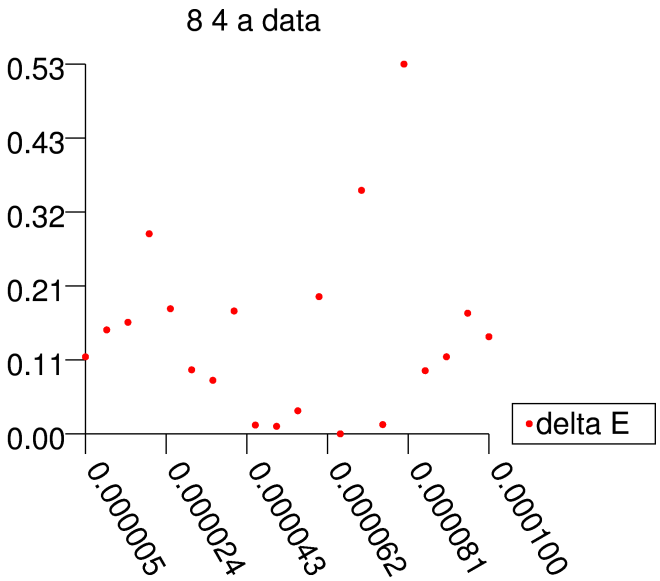
8.4. Tests of the Verlet algorithm

8.4.a. As was mentioned earlier, a timestep greater than .0001 showed instability under some conditions. Therefore, only timesteps less than .0001 will be compared. Specifically, we will compare the ability of the system to conserve total energy over a given amount of time. Due to the small timesteps being used, only .2 seconds will be simulated for each timestep. Since each configuration begins with a random distribution of atoms, the energy at the beginning will be discarded. The final average total energy will be compared with the average total energy after .1 seconds. This difference, ΔE, will be found for several timesteps. These tests were run using a common file to set the initial conditions. The results are in the chart below.

timestep	ΔE
.0001	0.140474202517
.000095	0.174189516177
.00009	0.111766784195
.000085	0.0918405748973
.00008	0.530936690316
.000075	0.0146528192887
.00007	0.350148387792
.000065	0.0013863317875
.00006	0.197912491535

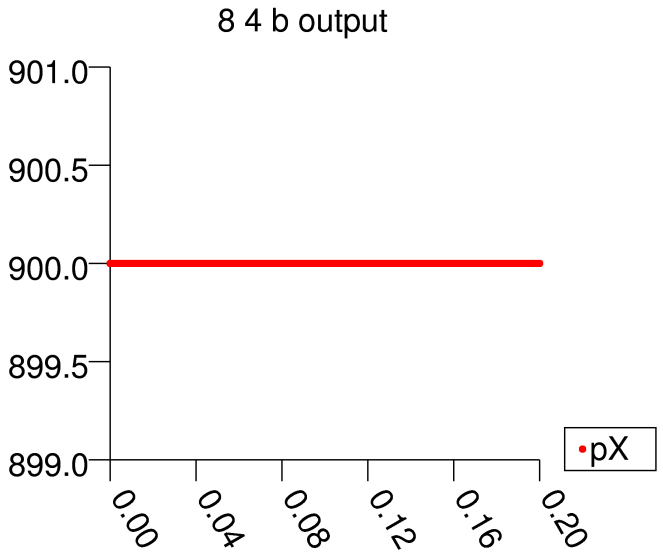
.000055	0.0343959021192
.00005	0.0121546426578
.000045	0.0139387086095
.00004	0.177327293246
.000035	0.0780376354455
.00003	0.0930706934455
.000025	0.180603398907
.00002	0.287946522817
.000015	0.161136139403
.00001	0.150275094578
.000005	0.111724465512

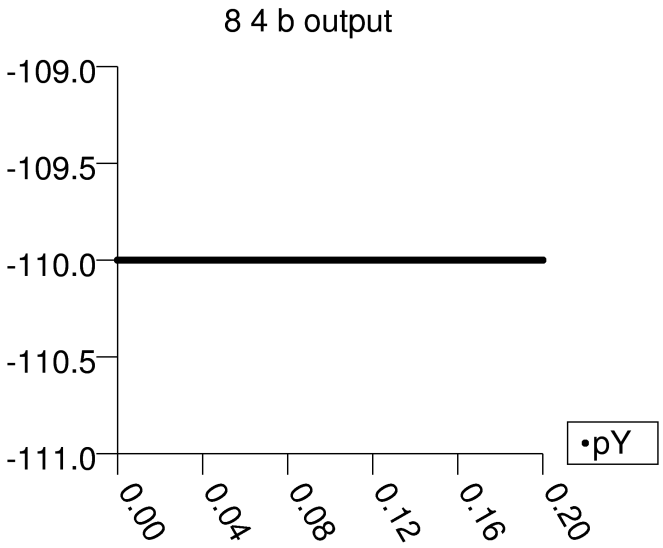
The graph below shows this data visually.:



These data appears to show no strong correlation between timestep size and the ability of the simulation to conserve total energy. Perhaps a correlation would have been found at timesteps greater than .0001. Such timesteps were not investigated because of their inherent instability.

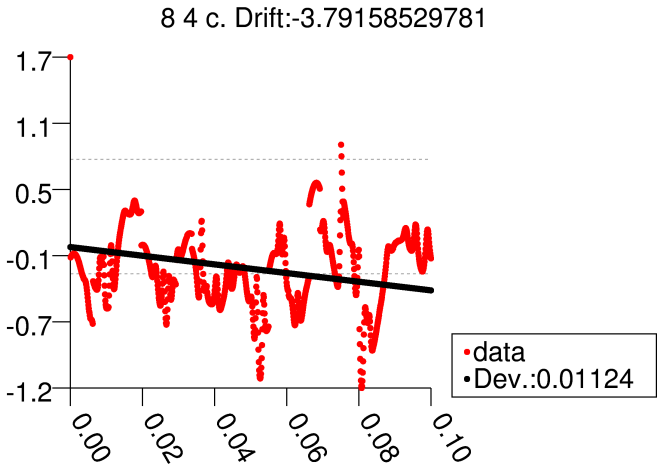
8.4.b. The ability of the simulator to conserve momentum is shown in two graphs below -- one for each dimension.



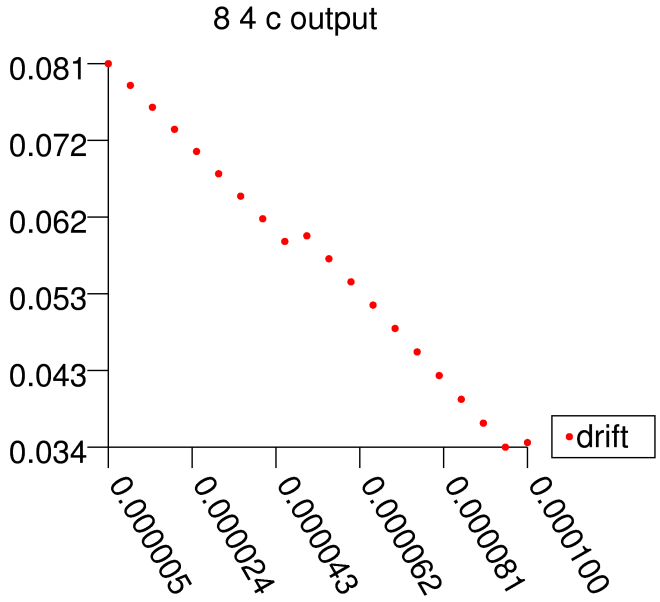


Apparently, this simulator does a very good job of preserving momentum. As can be seen above, the total linear momentum in the x-axis is 900, and -110 in the y-axis. These values are maintained during the simulation.

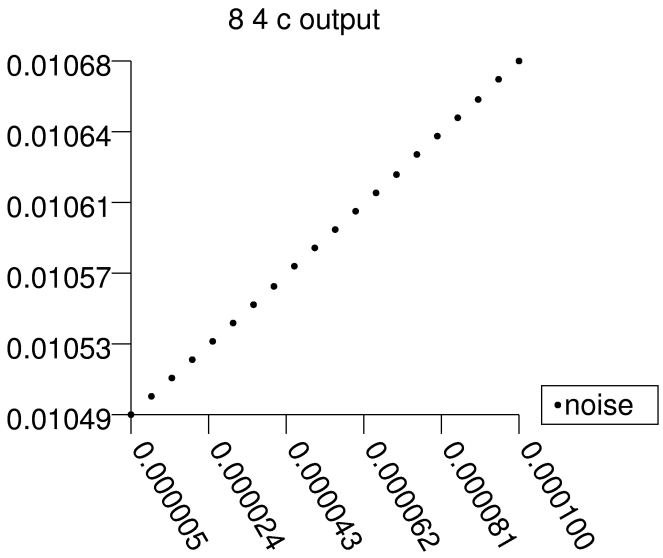
8.4.c. The total energy was plotted against time, and a line was fitted to the data minimizing the RMS error.



Using a timestep of .0001, the slope, or "drift" is -3.79, with a standard deviation, or "noise" of .01124. Results were found for other timesteps as well. Below is drift plotted against timestep.:

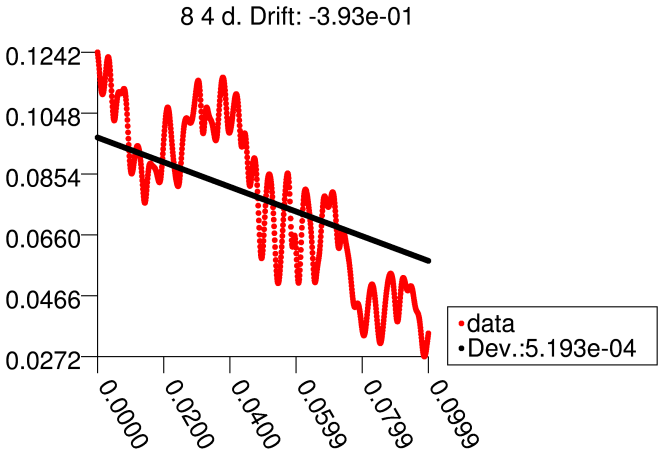


Next, below is noise against timestep.:



As the timestep increases it appears that the noise also increases. This is as would be expected. The unexpected correlation, however, was that there was less drift for higher timesteps.

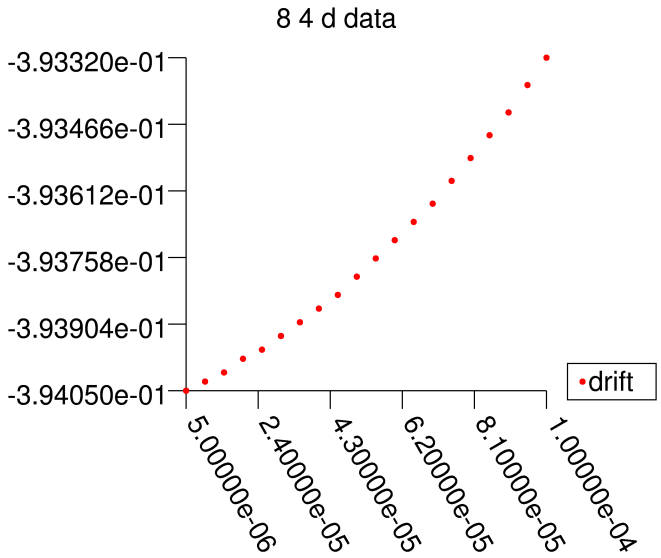
8.4.d. The Runge-Kutta method was used to perform a simulation with 16 atoms that had already reached somewhat of an equilibrium. Below is a chart of the total energy as a function of time with a timestep of .0001.

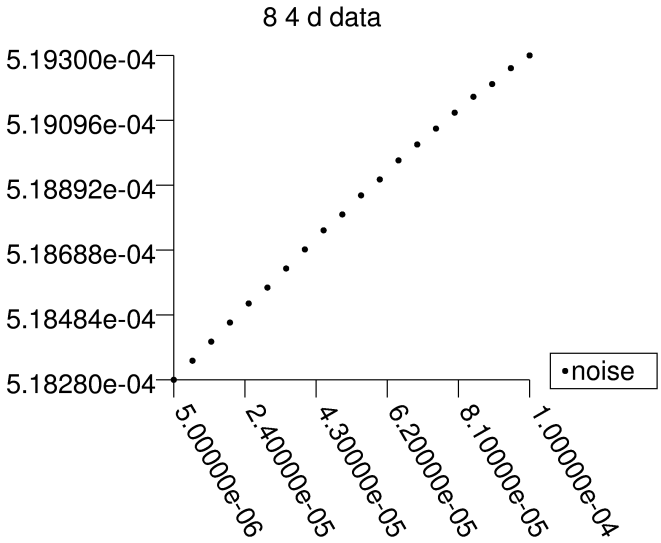


Already it can be seen that RK brings an improvement to the simulator. The same starting conditions were used to perform the same simulation for a number of other timesteps:

timestep	drift	deviation
0.000100	-3.9332e-01	5.1930e-04
0.000095	-3.9338e-01	5.1926e-04
0.000090	-3.9344e-01	5.1921e-04
0.000085	-3.9349e-01	5.1917e-04
0.000080	-3.9354e-01	5.1912e-04
0.000075	-3.9359e-01	5.1907e-04
0.000070	-3.9364e-01	5.1902e-04
0.000065	-3.9368e-01	5.1897e-04
0.000060	-3.9372e-01	5.1891e-04
0.000055	-3.9376e-01	5.1886e-04
0.000050	-3.9380e-01	5.1880e-04
0.000045	-3.9384e-01	5.1875e-04
0.000040	-3.9387e-01	5.1869e-04
0.000035	-3.9390e-01	5.1863e-04
0.000030	-3.9393e-01	5.1857e-04
0.000025	-3.9396e-01	5.1852e-04
0.000020	-3.9398e-01	5.1846e-04
0.000015	-3.9401e-01	5.1840e-04
0.000010	-3.9403e-01	5.1834e-04
0.000005	-3.9405e-01	5.1828e-04

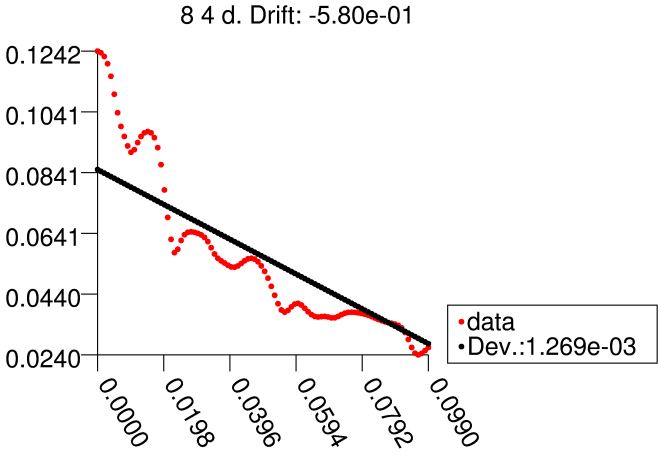
These data can be seen visually below:





The Runge-Kutta method (RK) produces simulations with much less drift and noise. The drift with RK is an order of magnitude smaller, and the noise is two orders of magnitude smaller. Here we also see that there is a strong correlation between drift, noise, and timestep. Decreasing the timestep lowers the noise and drift, as would be expected.

Using RK, it was possible to run with a larger timestep (.001) without experiencing the instabilities mentioned earlier. The total energy was also maintained better than it was under much smaller timesteps without RK. The drift was only -.580 with noise=.001269 when using dt=.001.



8.5 Qualitative properties of a liquid and a gas

8.5.a The temperature of the simulated system will be found by the following equation, applicable to two dimensions:

$$T = K / Nk$$

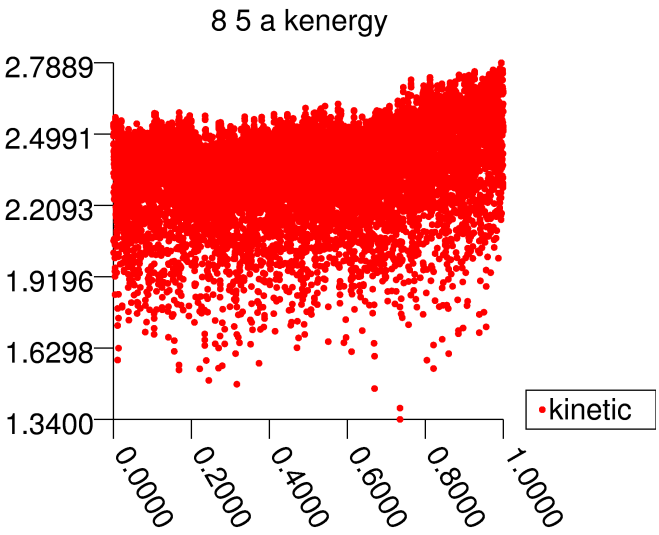
where K is the total kinetic energy, N is the number of particles, and k is the Boltzmann constant, which we will take to be 1 in our unitless world.

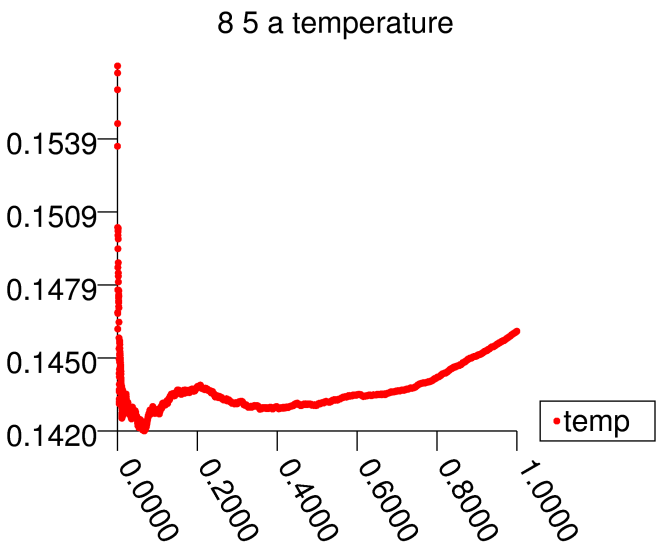
The pressure will be found similarly, from the equation:

$$P V = N R T$$

where we will take the constant R to be 1.

A simulation was run with 16 atoms in a volume of size 6X6. On the left is a graph of the kinetic energy over a timeperiod of 1s with a timestep of .0001. On the right is a graph of the ***time-averaged*** temperature over that same period.





The average temperature over this time period is 0.146039, which results in a pressure (according to the ideal gas law), of: $P = N T / V = 16*T/36 = 0.064906$.

To compensate for interactions between the atoms, the equation is ammended by adding the ***virial*** term to the definition of P.:

$$P = N T / V + (\sum_{ij} r_{ij} F_{ij}) / (d V)$$

where:

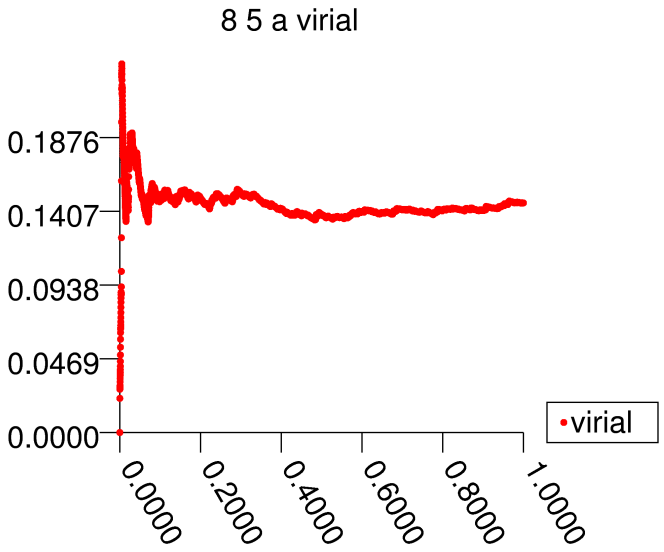
r_{ij} = distance vector between atoms i and j

F_{ij} = force vector between atoms i and j

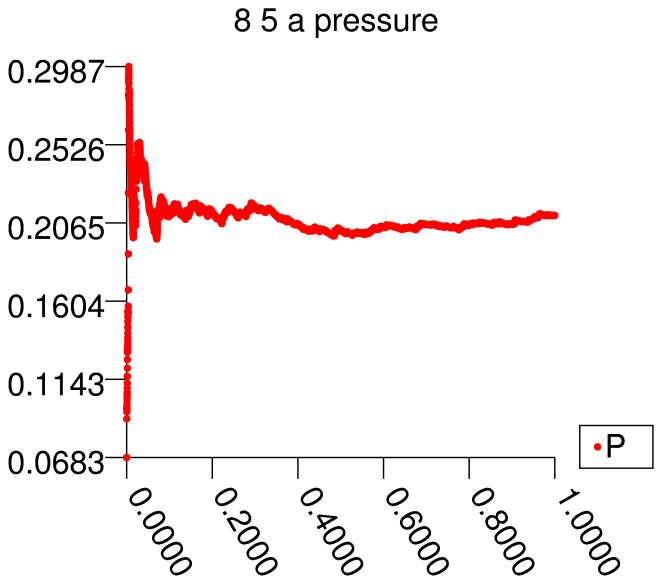
d = number of dimensions

V = volume

Below is a graph of the virial term over the same time period. The value has been normalized by a constant to be within the same magnitude as the temperature.



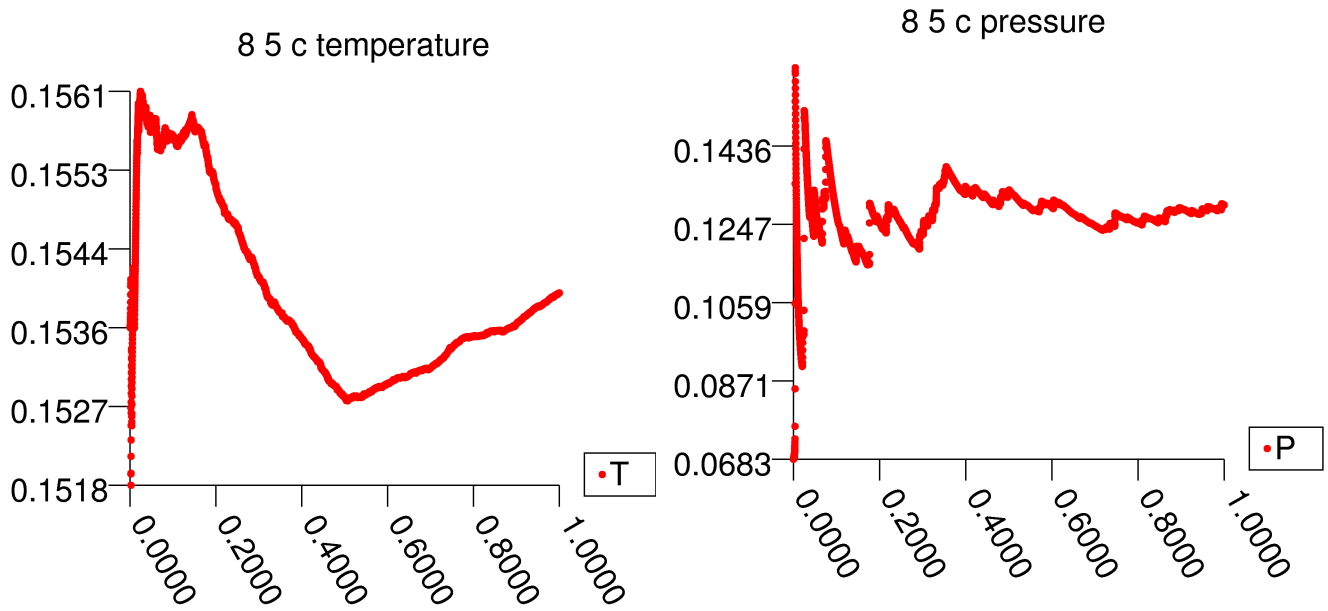
Then adding both terms we get a graph of pressure including the virial term.:



8.5.b. This step was effectively already done because the initialization file used in 8.5.a was generated by randomly distributing 16 atoms and saving their configuration once they had reached a relative equilibrium as understood by monitoring the drift of the total energy. The equilibrium was not perfect, but it was as good as this system appears to be able to maintain.

8.5.c. The simulation of 8.5.a was rerun with a slight alteration. All distances were increased by a factor of 2. I chose to increase the chamber size instead of decreasing it because of the unpredictable effects of putting two atoms too close together. It is expected that such a change should reduce the pressure considerably. All other factors included velocity were held constant.

Below left is a graph of the temperature of this modified system over the same time period. Below right is a graph of the pressure of this modified system over the same time period.

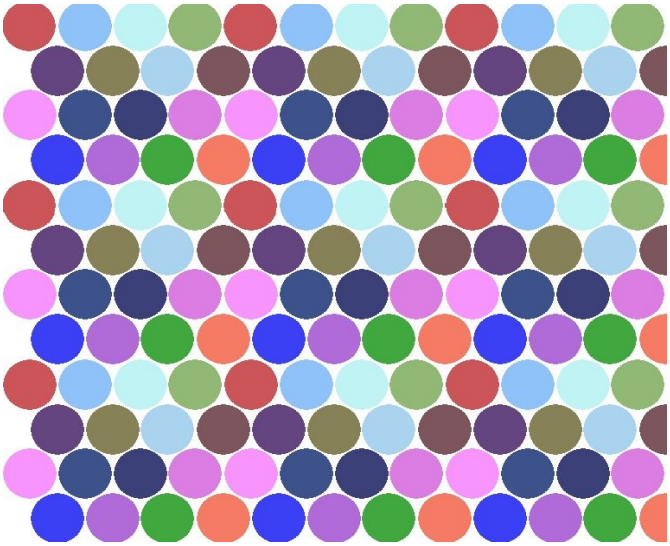


As can be seen above, the pressure is slightly more than half of the original pressure.

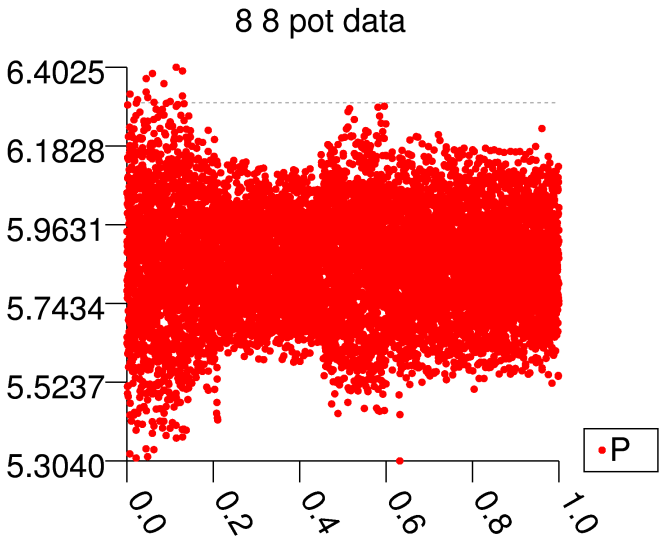
8.8. Two Dimensional Lattices

A simulation was performed using a triangular configuration of 16 atoms reflected into a total of nine cells. Considering a radius of 0.44544 and a central cell of size 3.635 X 3.125, this configuration had a density of 1.4084 atoms per unit volume.

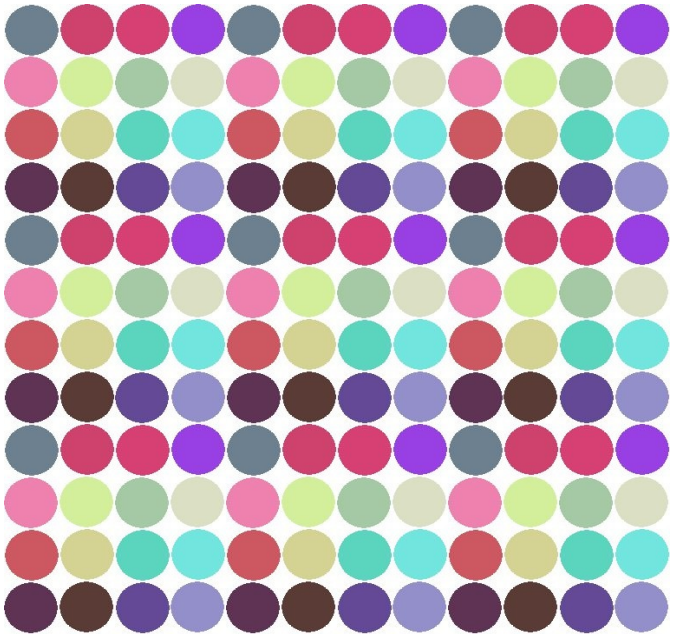
At first it was difficult to place all of the atoms without causing collisions in the initial state. A small epsilon was gradually modulated until a high density was reached while still having a small gap of .012 between each atom.



Below is the potential energy per particle as a function of time. The simulation was run over a period of 1 second with a timestep of .0001.

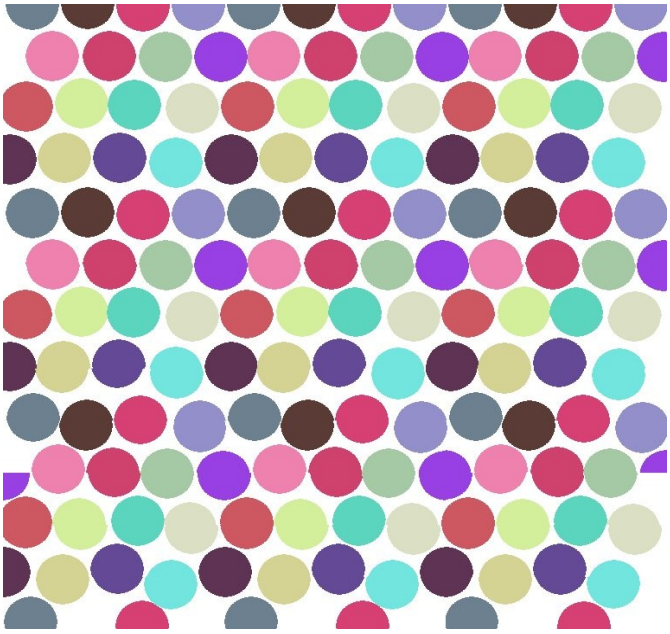


A similar simulation was done with a square lattice. 16 atoms were placed in a cell sized 3.564 X 3.564. In this case, the denisty of the lattice is 1.2595 atoms per unit volume. Below is a screenshot of this simulation near the start.

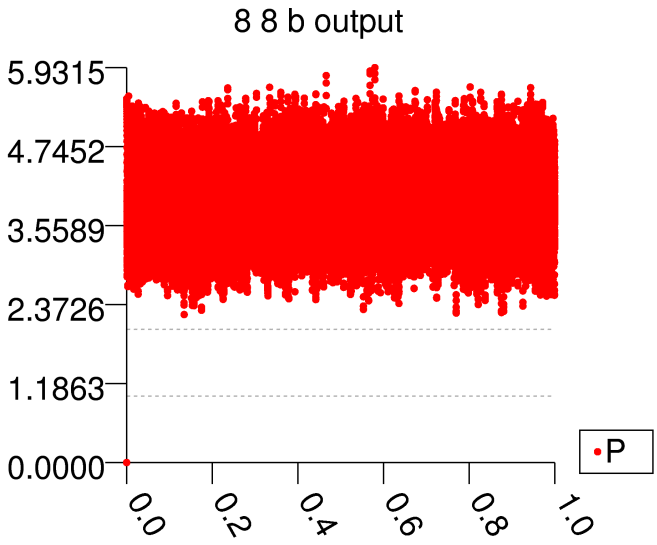


This scenario was not stable like the triangular lattice. A number of such simulations were run and the square lattice invariably decays into a triangular lattice. Below is a screenshot of the simulation just after the square lattice began to break up.

After a very short time of appearing somewhat chaotic, the simulation settles into a triangular lattice, as shown below.



Below is the potential energy per particle as a function of time. The simulation was run over a period of 1 second with a timestep of .00001. The reason for the smaller timestep was the greater propensity for extreme behavior. Although both of the lattices share a high potential energy, the instability of the square lattice results in a likelihood that some atoms will eventually have a high kinetic energy -- a situation that can produce numbers higher than the system maxfloat.

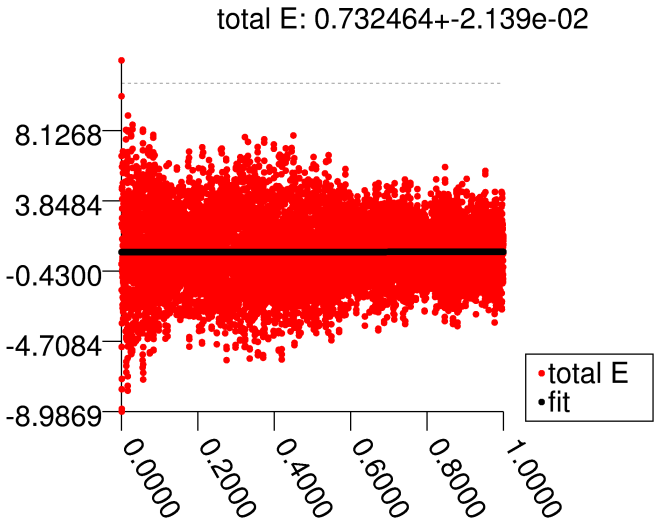


The potential energy per particle is significantly less in the square lattice, an obvious result of having each atom as close as possible to four others, as opposed to six other atoms.

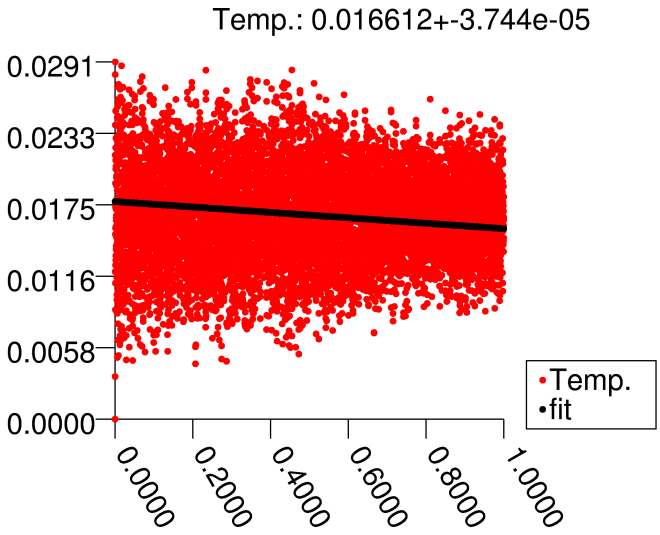
Besides the obvious point that the triangular lattice appears more dense, the density numbers above show that it was indeed more dense, assuming that both configurations were packed as densely as reasonably possible.

8.9. Solid state and melting

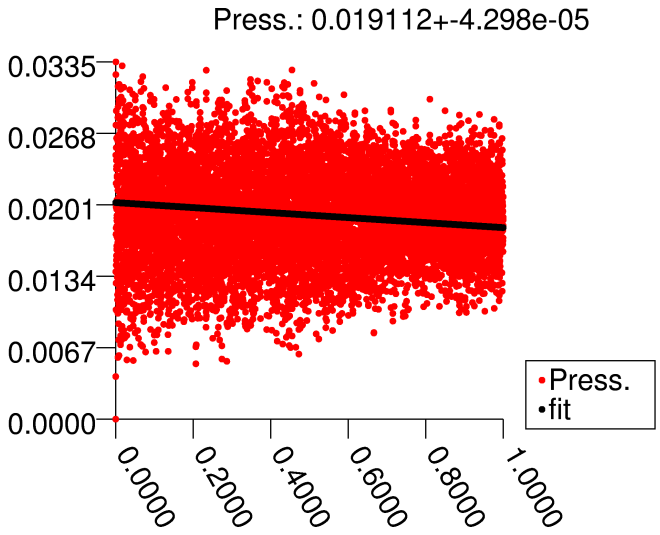
8.9.a. A triangular lattice was started with 16 atoms in a cell sized 4.01 X 3.46. Each atom started with a velocity of zero. The simulation was run for 1 second with a timestep of .0001. The following is the total energy of the system during that time period. The mean energy was .732 ±.002 (std. dev.).



Below is the temperature over the same time period. The mean temperature was .0166 ±4e-5.:

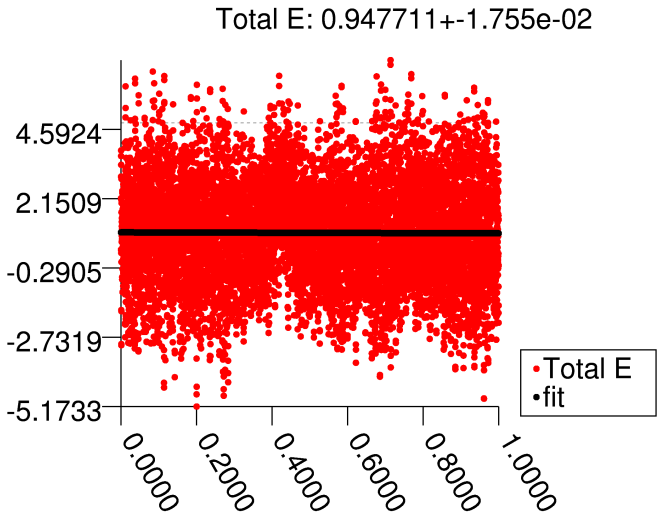


Below is the pressure over the same time period. The mean pressure was .019 ±4e-5.:

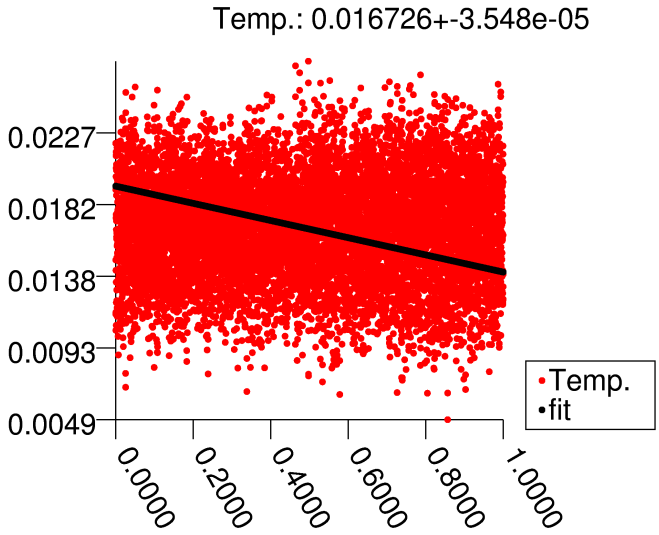


The simulation shows that the system does indeed remain a solid. The simulation was run for more than 50 seconds, and each atom stayed in the same place in the lattice.

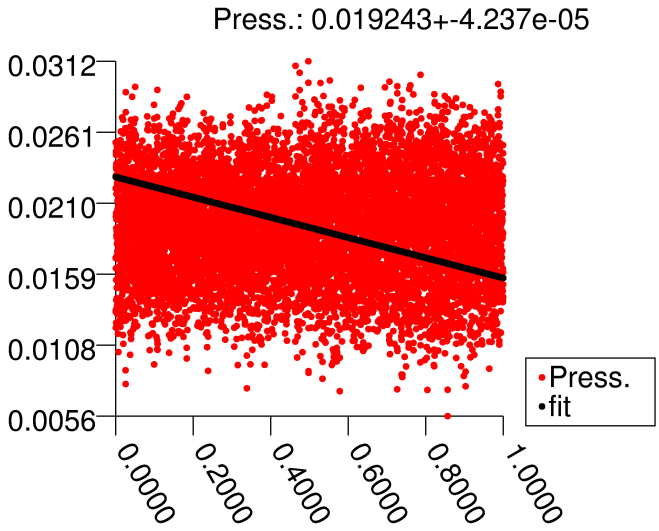
8.9.b. Each particle was given a random velocity. The velocities were between -100 and +100, and then the system was allowed to stabilize for 1 second. A snapshot was taken at that point and this snapshot was the initial configuration for this simulation. The graph below shows the total energy over a period of one second with a timestep of .0001. The mean of the total energy was .948 ±.018



The graph below shows the temperature over the same time period. The mean temperature was .01673 ±4e-5. The fitted line appears to be askew. It is not clear why the angle of the line doesn't match the intuitive slope of the graph, but this angle was determined to have the lowest RMS error after modulating the slope and intercept significantly with finer and finer tuning. Numerous attempts were made to debug the line fitting feature.



Below is a graph of the pressure in this system over the same time period. The same angle can be seen in this graph as well. The mean pressure was .01924 ±4e-5.



Surprisingly, the atoms do indeed remain a solid -- each atom remains in its place in the triangular lattice. It appears to be a remarkably stable configuration.

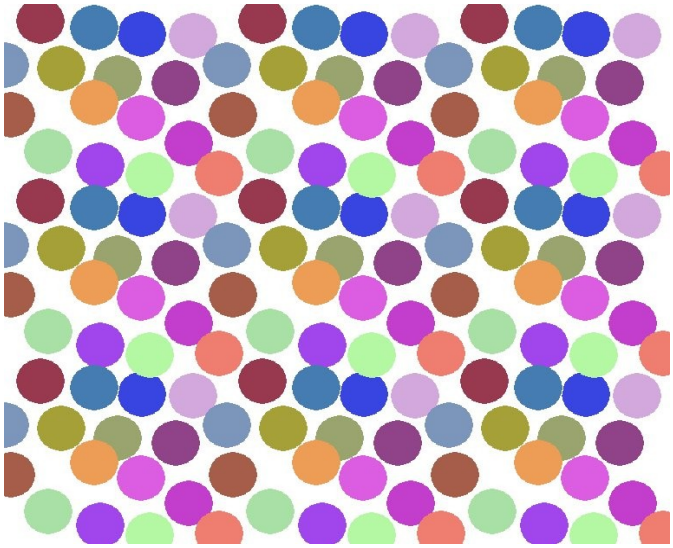
8.9.c. In this simulation the kinetic energy was increased. The same initial conditions were used, but the velocity of each atom was scaled up before starting.

In the first simulation, the velocity was doubled in both dimensions. This results in a 4-fold increase in the kinetic energy. Remarkably, the atoms remain in a solid, but their trajectories look much more chaotic. Despite the extra activity, there is still a visually identifiable triangular lattice, and each atom remains in place.

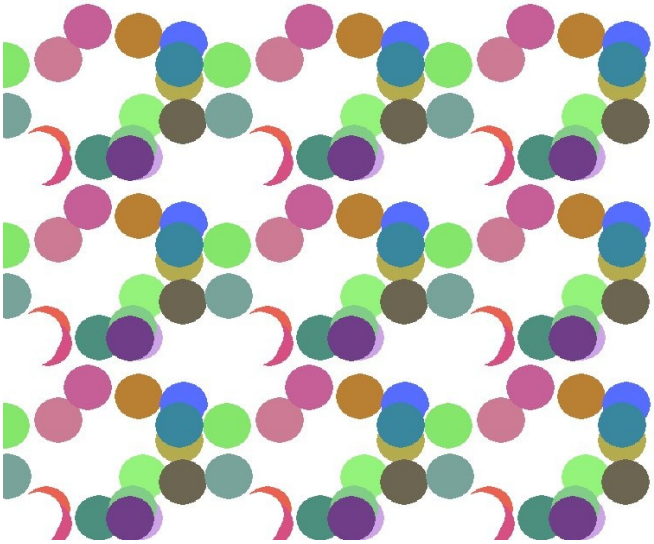
The second simulation multiplied the initial veloicities by 3 -- same qualititative result with a slightly higher level of excitation.

The third simulation, multiplying the velocities by 4, however, very quickly devolved into something non-solid. From the beginning the atoms looked extremely agitated.

At .0115 seconds their agitation reached a breaking point.:



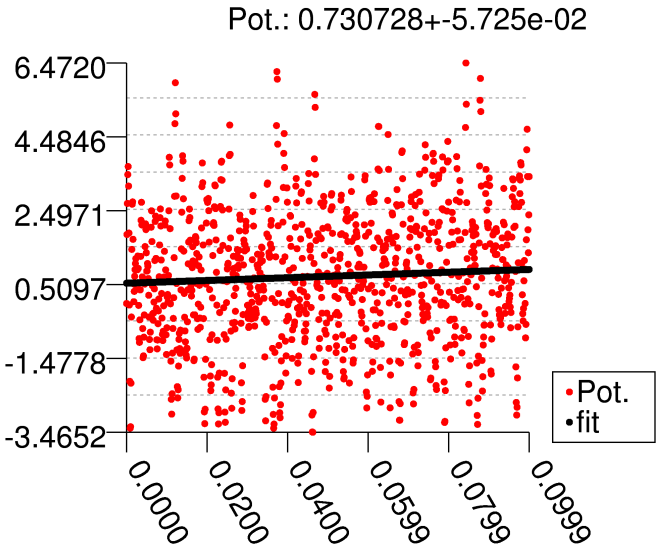
Then, at .0117 seconds their lattice totally disolved and liquefaction resulted.:



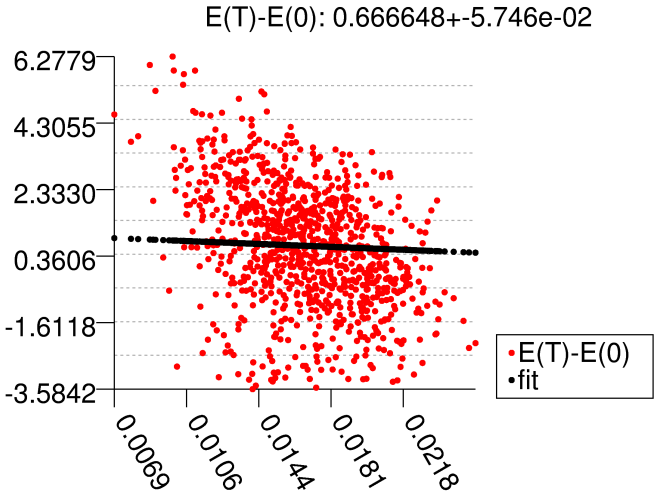
As can be seen, the drawing mechanism was not able to keep up with the moving atoms. The white circles are what the animation code uses to erase the previous location of an atom.

These chaotic trajectories do not appear to be a simulation of melting. Perhaps it is similar to a highly pressurized gas, or plasma(not likely - as each atom is being modeled as a whole). It is more likely that at these energy levels the simulation is no longer valid.

8.9.d. The mean potential energy for the solid being simulated is approximately .73 ±.06.

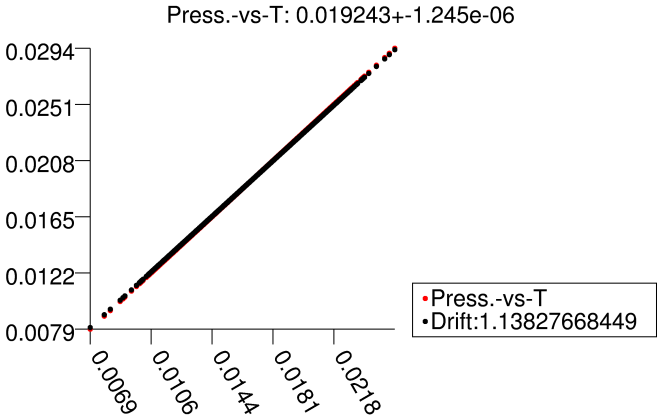


The below graph shows the relationship between E(T)-E(0) and temperature T. This is another situation where the fitted line doesn't match where I would intuitively draw the line. I can't explain it except to say that I check the fitting process several times and keep getting this same result.



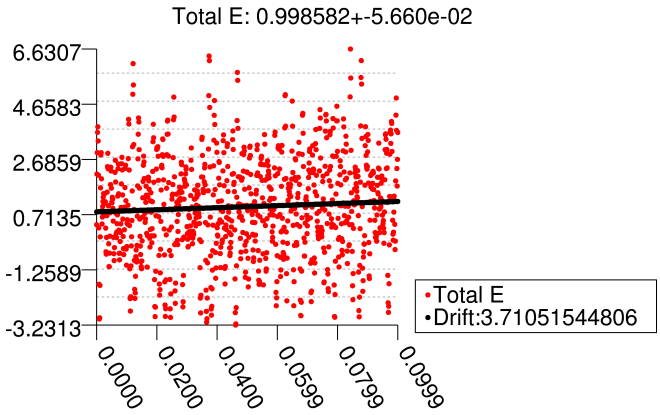
It appears that there is no strong relationship between E(T)-E(0) and temperature.

Below is a graph of the pressure plotted against T.

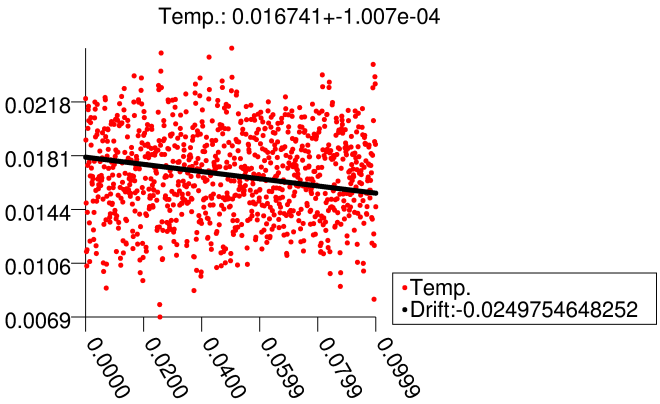


This exact correlation comes from the fact that Pressure relates directly to temperature.

Below is a graph of the total energy over the time period of the simulation. The drift was approximately 3.7105.

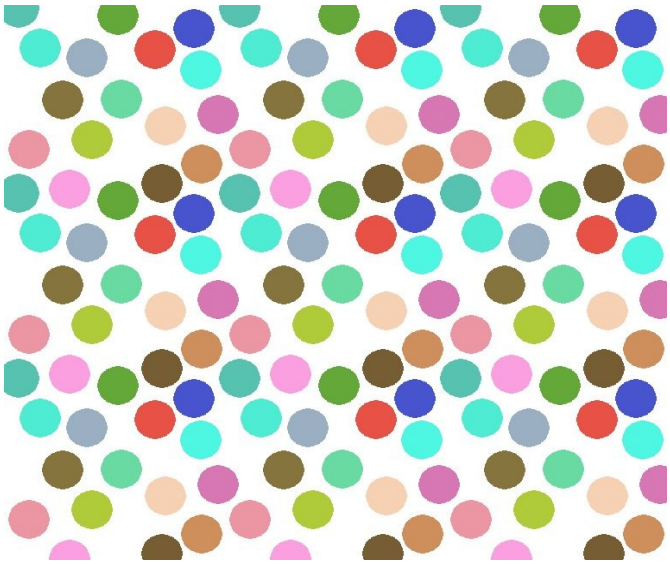


Below is a graph of the temperature of the same time period. The drift was approximately .024975.



The heat capacity is therefore 3.7105 / .024975 = - 148.6

8.9.e. The density of the system was decreased by increasing the size of the cell slightly until melting occured. A number of different simulations were run. Melting was visually confirmed once the density reached approximately .85 atoms per unit of volume. Melting occurs when there is no identifiable consistent lattice. There may be moments were something that appears to be a lattice appears, but no configuration remains.



CONCLUSION

The system outlined above proved to be highly flexible and useful for running many disparate tests. This method of modeling appears to be robust under many conditions and is visually very interesting. The fact that quantities such as kinetic energy, potential, and temperature, can be monitored with such precision means that these simulated atoms can be studied in ways that real atoms never could be.

BIBLIOGRAPHY

[1] Wikipedia article, "Lennard–Jones potential"
http://en.wikipedia.org/wiki/Lennard-Jones_potential

[2] H. Gould, et al. "An Introduction to Computer Simulation Methods"
Chapter 8

Appendix: CODE

The below code was originally stored is several files but is listed in two files here for simplicity. The first file is "atom.py," the main file used to run simulations. The second set of code is everything else that was used to either support the simulation or plot graphs, etc.

The numerous experiments with slight alterations were sometimes performed by commenting out alternate versions of commands instead of creating new functions. This cut down on the overall length of the code, but it should be noted that most commented out lines were uncommented at some point.

FILE: atom.py :

```
##  Globals
global keFactor
keFactor = 13e-7


##################
##  Atom CLASS  ##
##################

class Atom:
    ''' Models a single atom  '''
    # Constructor
    def __init__(self, cell, x, y, xDot=0, yDot=0, m=1):
        self.cell = cell
        self.radius = cell.radius
        self.x = x
        self.y = y
        if self.x > cell.Lx:
            print 'ERROR: x too big:',self.x
            raise Exception('xTooLarge')
        if self.y > cell.Lx:
            print 'ERROR: y too big:',self.y
            raise Exception('yTooLarge')

        self.xDot = xDot
        self.yDot = yDot
        self.m = m
        self.color = randint(50,255), randint(50,255), randint(50,255)    # random color
        self.pot = 0    # potential energy
        self.force = [0,0]


    def animate(self):
        if self.x < self.cell.Lx and self.y < self.cell.Lx:
            self.circle = cell.anim.mkCircle([self.x, self.y], self.radius, self.color)
        else:
            self.cell.disown(self)


    def delete(self):
        cell.anim.mkCircle([self.x, self.y], self.radius, [255,255,255])
        self.cell.disown(self)


    def distance(self, other):
        d = 0
        d += (other.y - self.y)**2
        d += (other.x - self.x)**2
        dist = math.sqrt(d)
```

```
    return dist


def distanceXY(self, x, y):

    d = 0
    d += (y - self.y)**2
    d += (x - self.x)**2
    dist = math.sqrt(d)

    return dist


def get_potential(self, other):
    ''' Compute a unitless potential energy between two atoms '''
    pot = 0
    r = self.distance(other)
    if r > self.cell.distant:
        pass
    else:
        r6 = r**6
        pot = (1/r6 - 1)/ r6

    self.pot += pot
    other.pot += pot


def get_xyDot(self, x, y, a=None, b=None):
    ''' Compute x,y Dot from the derivative of the LJ potential '''
    r = None
    if a==None or b==None:
        a = self.x
        b = self.y

    r = distance([x,y], [a,b])

    if r > self.cell.distant:
        rDot = 0
    elif r == 0:
        return 0,0
    else:
        r6 = r**6
        rDot = (self.cell.coeff/(r*r6)) * (1 - 2*self.cell.sigma6/r6) - self.cell.offset

    theta = math.atan2(y-b, x-a)
    delta_xDot = rDot*math.cos(theta)
    delta_yDot = rDot*math.sin(theta)
    return delta_xDot, delta_yDot


def getAll_xyDot(self, x, y, atoms=None):
    ''' Compute x,y Dot from the derivative of the LJ potential '''
    if atoms == None:
        atoms = self.cell.atoms
    g = 10000 * self.cell.step
    delta_xDot = 0
    delta_yDot = 0
    for a in atoms:
        if a.x > self.x:
            Lx = -self.cell.Lx
        else:
            Lx = self.cell.Lx
        if a.y > self.y:
            Ly = -self.cell.Ly
        else:
            Ly = self.cell.Ly

        d_xDot, d_yDot = self.get_xyDot(a.x, a.y, x, y)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
        d_xDot, d_yDot = self.get_xyDot(a.x+Lx, a.y, x, y)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
```

```
        d_xDot, d_yDot = self.get_xyDot(a.x, a.y+Ly, x, y)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
        d_xDot, d_yDot = self.get_xyDot(a.x+Lx, a.y+Ly, x, y)
        delta_xDot += d_xDot
        delta_yDot += d_yDot

    return g*delta_xDot, g*delta_yDot


# XPSYS
def xpsys(self, Q, atoms):
    F = []

    delta_xDot, delta_yDot = self.getAll_xyDot(Q[1], Q[3], atoms)

    # t
    F.append(1)

    # x
    F.append(Q[2])

    # xDot
    F.append(delta_xDot)

    # y
    F.append(Q[4])

    # yDot
    F.append(delta_yDot)

    return F


# RK4SYS
def RK4SYS(self, Q, atoms):
    h = self.cell.step
    F1 = self.xpsys(Q, atoms)
    Y1 = sumLists([Q, multList(h, F1)])

    F2 = self.xpsys(Y1, atoms)
    Y2 = sumLists([Q, multList(h, F2)])

    F3 = self.xpsys(Y2, atoms)
    Y3 = sumLists([Q, multList(2*h, F3)])

    F4 = self.xpsys(Y3, atoms)

    return sumLists([Q, multList(h/3, sumLists([F1, multList(2, sumLists([F2, F3])), F4]))])


def updateRK(self):
    # Q vector:
    #     0    t
    #     1    x
    #     2    xDot
    #     3    y
    #     4    yDot
    others = []
    for a in self.cell.atoms:
        if not a==self:
            others.append(a)

    Q = [self.cell.time0-self.cell.time, self.x, self.xDot, self.y, self.yDot]
    Q = self.RK4SYS(Q, others)
    self.xDot = Q[2]
    self.yDot = Q[4]


def updateVelocity(self, other):
    if other.x > self.x:
        Lx = -self.cell.Lx
    else:
        Lx = self.cell.Lx
```

```
        if other.y > self.y:
            Ly = -self.cell.Ly
        else:
            Ly = self.cell.Ly

        # Compute for atom and three nearest reflections
        delta_xDot = delta_yDot = 0
        d_xDot, d_yDot = self.get_xyDot(other.x, other.y)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
        force = [d_xDot, d_yDot]

        d_xDot, d_yDot = self.get_xyDot(other.x+Lx, other.y)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
        force = [force[0]+d_xDot, force[1]+d_yDot]

        d_xDot, d_yDot = self.get_xyDot(other.x, other.y+Ly)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
        force = [force[0]+d_xDot, force[1]+d_yDot]

        d_xDot, d_yDot = self.get_xyDot(other.x+Lx, other.y+Ly)
        delta_xDot += d_xDot
        delta_yDot += d_yDot
        force = [force[0]+d_xDot, force[1]+d_yDot]

        self.force = force
        other.force = force
        self.xDot += delta_xDot
        self.yDot += delta_yDot
        other.xDot -= delta_xDot
        other.yDot -= delta_yDot


    def updatePosition(self):
        self.x += self.xDot * self.cell.step
        self.y += self.yDot * self.cell.step

        """
        if self.x > self.cell.Lx:
            self.x = self.x % self.cell.Lx
        elif self.x < 0:
            self.x = self.x % self.cell.Lx

        if self.y > self.cell.Ly:
            self.y = self.y % self.cell.Ly
        elif self.y < 0:
            self.y = self.y % self.cell.Ly
        """
        self.x = self.x % self.cell.Lx
        self.y = self.y % self.cell.Ly

        if self.cell.nodisplay:
            return
        try:
            self.circle.update({'position':(self.x, self.y), 'screen':self.cell.anim.screen})
        except Exception, e:
            print 'ERROR in updatePosition:',e
            print '\t with atom:',self
            self.cell.menu()


    def collision(self, other):
        if self.distance(other) <= 2*self.radius:
            return True
        return False

    def collisionXY(self, x, y):
        if x > self.x:
            Lx = -self.cell.Lx
        else:
            Lx = self.cell.Lx
        if y > self.y:
```

```
            Ly = -self.cell.Ly
        else:
            Ly = self.cell.Ly

        r = 2*self.radius
        if self.distanceXY(x, y) <= r:
            return True
        if self.distanceXY(x+Lx, y) <= r:
            return True
        if self.distanceXY(x, y+Ly) <= r:
            return True
        if self.distanceXY(x+Lx, y+Ly) <= r:
            return True
        return False



##################
##  Cell CLASS  ##
##################

class Cell:
    ''' Environment for a set of atoms  '''
    # Constructor
    def __init__(self, options={'size':10, 'cells':1, 'step':.1}, epsilon=1, sigma=1):
        self.Lx = options['size']
        if options.has_key('sizeY'):
            self.Ly = options['sizeY']
        else:
            self.Ly = options['size']  # area usually a square
        self.step = options['step']
        self.time = options['time']
        self.time0 = self.time
        self.cells = options['cells']
        self.epsilon = epsilon
        self.sigma = sigma
        self.sigma6 = sigma**6    # do exp once to save computation time
        self.coeff = 24 * self.epsilon * self.sigma6
        self.atoms = []
        self.radius = .5 * 0.89089871814033927 / sigma
        self.distant = 2.5*self.sigma
        self.pairs_atoms = []
        if options.has_key('nodisplay'):
            self.nodisplay = options['nodisplay']
        else:
            self.nodisplay = False
        try:
            self.update_extra
        except:
            self.update_extra = None
        try:
            self.gotoMenu = options['menu']
        except:
            self.gotoMenu = True
        if options.has_key('rk'):
            self.rk = options['rk']
        else:
            self.rk = False

        r6 = (self.distant)**6
        self.offset = (self.coeff/(self.distant*r6)) * (1 - 2*self.sigma6/r6)


    def collision(self, x, y):
        for atom in self.atoms:
            if atom.collisionXY(x,y):
                return True
            if atom.x > self.Lx or atom.y > self.Ly or \
                    atom.x < 0 or atom.y < 0:
                return True
        return False


    def randomAtoms(self, n):
```

```
    for a in range(n):
        self.addAtom()


def addAtom(self, x=None, y=None, xDot=None, yDot=None, m=1):
    if x==None or y==None:
        x = random()*self.Lx
        y = random()*self.Ly
        tries = 0
        while self.collision(x,y) and tries < 50:
            tries += 1
            x = random()*self.Lx
            y = random()*self.Ly
        if tries >= 50:
            return False
    else:
        if self.collision(x,y):
            return False

    if xDot==None:
        xDot = 800*(2*random()-1)
    if yDot==None:
        yDot = 800*(2*random()-1)

    try:
        a = Atom(self, x, y, xDot, yDot, m)
        self.atoms.append(a)
        return True
    except:
        return False


def disown(self, atom):
    i = -1
    for a in self.atoms:
        i += 1
        if a == atom:
            del self.atoms[i]


def pairWithEach(self, a, atoms):
    pairs = []
    for b in atoms:
        pairs.append([a,b])
    return pairs


def pairs(self):
    if len(self.pairs_atoms) > 0:
        return self.pairs_atoms
    pairs = []
    for i in range(len(self.atoms)):
        pairs.extend(self.pairWithEach(self.atoms[i], self.atoms[i+1:]))
    self.pairs_atoms = pairs
    return self.pairs_atoms


def update(self):
    if not self.update_extra == None:
        self.update_extra(self)

    # monitor time
    self.time -= self.step
    # print self.time
    if self.time <= 0:
        try:
            self.save(self.saveConfig)
        except:  pass
        if not self.menu():
            return False

    if self.rk:
        for a in self.atoms:
            a.updateRK()
```

```
            for a in self.atoms:
                a.updatePosition()
            return True

        else:
            A = self.atoms[0]
            origXdot = A.xDot
            origYdot = A.yDot

            for a,b in self.pairs():
                a.updateVelocity(b)
            for a in self.atoms:
                a.pot = 0
            for a,b in self.pairs():
                a.get_potential(b)

            for a in self.atoms:
                a.updatePosition()
            return True


    def alterSize(self, add):
        if self.Lx + add < 2 or self.Ly + add < 2:
            return False
        self.Lx += add
        self.Ly += add
        self.quit()
        atoms = self.atoms
        self.atoms = []
        for a in atoms:
            self.addAtom(a.x, a.y, a.xDot, a.yDot)
        self.animate()


    def handle(self, key):
        if key == None:
            pass

        # (m) menu
        elif key == 109:
            self.quit()
            self.menu()

        # (s) scale - up
        elif key == 115:
            self.alterSize(1)
        # (x) scale - down
        elif key == 120:
            self.alterSize(-1)

        # density

        else:
            print key


    def animate(self):
        self.anim = Animation(self.Lx, self.Ly, self.cells, {'nodisplay':self.nodisplay})
        for a in self.atoms:
            a.animate()

        while True:
            try:
                self.anim.draw()
            except Exception, e:
                print 'ERROR:',e
                self.anim.quit()
                exit()
            self.handle(self.anim.checkEvent())

            if not self.update():
                return False
```

```
def quit(self):
    self.anim.quit()


def printState(self):
    print '\nState:'
    print '\tsize:',self.Lx
    print '\tcells:',self.cells
    print '\tstep:',self.step
    print '\ttime:',self.time
    print


def setup(self, num=None, choice=None):

    if num == None:
        print 'Number of atoms:'
        num = int(sys.stdin.readline()[:-1])
        print '(r)andom,  (t)riangular, or  (s)quare:'
    self.atoms = []

    if choice==None:
        choice = sys.stdin.readline()[:-1]

    # random
    if choice == 'r':
        self.randomAtoms(num)

    # square lattice
    elif choice == 's':
        mult = 2
        num = (2 * mult)**2
        n = 0
        r = self.radius
        e = .0001

        # set appropriate size
        self.Lx = mult * (4*r + 3*e)
        self.Ly = mult * (4*r + 3*e)
        #print 'SIZE:',self.Lx, self.Ly
        #print 'Radius:',self.radius
        #print

        dX = 2*r + e
        x = -dX/2

        while x+dX < self.Lx and n < num:
            x += dX
            dY = 2*r + e
            y = -dY/2
            while y+dY < self.Ly and n < num:
                y += dY
                xDot = 10*(2*random()-1)
                yDot = 10*(2*random()-1)

                if self.addAtom(x, y, xDot, yDot):
                    n += 1
                else:
                    print 'FAIL:',x,y


    # triangular lattice
    elif choice == 't':
        mult = 2
        num = (2 * mult)**2
        n = 0
        r = self.radius
        e = .075
        eY = -.07
        sin60 = math.sin(math.radians(60))
        cos60 = math.cos(math.radians(60))

        # set appropriate size
        self.Lx = mult * (4*r + 3*e)
```

```
        self.Ly = mult * (2*r + 2*r*sin60 - eY)
        #print 'SIZE:',self.Lx, self.Ly
        #print 'Radius:',self.radius
        #print

        dX = 2*r + e
        x = -dX/2
        shift = dX/2

        while x+dX < self.Lx and n < num:
            x += dX
            dY = 2 * self.radius * math.sin(math.radians(60)) + e
            y = -dY/2
            while y+dY < self.Ly and n < num:
                if shift == 0:
                    shift = dX/2
                else:
                    shift = 0
                y += dY

                xDot = 10*(2*random()-1)
                yDot = 10*(2*random()-1)

                if self.addAtom(x+shift, y, xDot, yDot):
                    n += 1
                else:
                    print 'FAIL:',x,y


def menu(self):
    if not self.gotoMenu:
        return False

    self.printState()
    if len(self.atoms) > 0:
        print '(s)tart,  (a)lter,  (c)ontinue,  (r)everse,  or  (q)uit?'
    else:
        print '(s)tart,  (a)lter,  or  (q)uit?'

    choice = sys.stdin.readline()[:-1]
    if choice == 'q':
        exit()
    elif choice == 'c':
        self.animate()
    elif choice == 'r':
        self.time = 10
        for a in self.atoms:
            a.xDot = -a.xDot
            a.yDot = -a.yDot
        self.animate()
    elif choice == 'a':
        print 'Change the state:'
        print '\t1. size'
        print '\t2. cells'
        print '\t3. step'
        print '\t4. time'
        print
        choice = int(sys.stdin.readline()[:-1])
        if choice == 1:
            print 'Enter new size:'
            self.size = int(sys.stdin.readline()[:-1])
        elif choice == 2:
            print 'Enter new number of cells:'
            self.cells = int(sys.stdin.readline()[:-1])
        elif choice == 3:
            print 'Enter new step:'
            self.step = float(sys.stdin.readline()[:-1])
        elif choice == 4:
            print 'Enter new time:'
            self.time = float(sys.stdin.readline()[:-1])
        self.menu()
    else:
        self.setup()
```

```
        self.animate()
        return True


    def save(self, file):
        f = open(file, 'w')

        print >>f, '%s\t%f,%f'% ('size', self.Lx, self.Ly)
        print >>f, '%s\t%d'% ('cells', self.cells)
        print >>f, '%s\t%f'% ('time', self.time0)
        print >>f, '%s\t%f'% ('step', self.step)

        for a in self.atoms:
            print >>f, '%s\t%f\t%f\t%f\t%f'% ('atom', a.x, a.y, a.xDot, a.yDot)


    def kineticEnergy(self):
        ''' Returns total kinetic energy '''
        global keFactor
        k = 0
        for a in self.atoms:
            k += keFactor*(a.xDot**2 + a.yDot**2)
        return k


    def pressure(self, T=None):
        N = len(self.atoms)
        if T == None:
            T = self.kineticEnergy()/N
        V = self.Lx * self.Ly
        P = N*T/V
        return P



##################
##  FUNCTIONS  ##
##################

def distance(a, b):
    ''' Generalized euclidian distance function  '''
    d = 0
    for i in range(len(a)):
        d += (a[i] - b[i])**2
    return math.sqrt(d)


def load(file, inoptions={}):
    options = {'size':10, 'cells':1, 'step':.0001, 'time':.1, 'time0':.1}
    scale = 1    # rescaling factor
    scaleV = 1    # Velocity rescaling factor
    for k in inoptions.keys():
        if k == 'rescale':
            scale = float(inoptions[k])
        elif k == 'rescaleV':
            scaleV = float(inoptions[k])
        else:
            options[k] = inoptions[k]

    f = open(file)
    cell = Cell(options)
    for line in f:
        line = line.strip()
        p = line.split('\t')
        if p[0] == 'size':
            s = p[1].split(',')
            cell.Lx, cell.Ly = scale*float(s[0]), scale*float(s[1])
        elif p[0] == 'cells':
            cell.cells = int(p[1])
        elif p[0] == 'step':
            cell.step = float(p[1])
        elif p[0] == 'time':
            cell.time = float(p[1])
            cell.time0 = cell.time
```

```
        elif p[0] == 'atom':
            x = scale * float(p[1])
            y = p[2]
            if re.search('/',y):
                n,d = y.split('/')
                y = scale*float(n)/float(d)
            else:
                y = scale* float(p[2])
            xDot = scaleV* float(p[3])
            yDot = scaleV* float(p[4])
            cell.addAtom(x, y, xDot, yDot)

    f.close()
    return cell


def printEnergy(cell):
    global totalP, totalK, obs, e0, e1
    obs += 1
    p = 0
    k = 0
    for a in cell.atoms:
        k += 13e-7*(a.xDot**2 + a.yDot**2)
        p += a.pot

    totalP += p
    totalK += k
    avgP = totalP/obs
    avgK = totalK/obs
    avg = avgP + avgK

    # use this clause for problem 8.4.a
    if .1-cell.step < cell.time <= .1:
        e0 = avg+avgK
    elif 0 <= cell.time < cell.step:
        e1 = avg+avgK
        print 'delta(',e0,',',e1,'):  ',abs(e0 - e1)
    print >>cell.fh, '%f\t%f\t%f\t%f\t%d'% (cell.time0-cell.time, avgP, avgK, avg, obs)
    # print cell.time0-cell.time, '\t', avgP, '\t', avgK


def printEnergy2(cell):
    global data
    p = 0
    k = 0
    # c = 13e-7    # coefficient to make p,k of same order of magnitude
    c = 4e-6
    for a in cell.atoms:
        k += c*(a.xDot**2 + a.yDot**2)
        p += a.pot

    data[cell.time0-cell.time] = p+k
    # print cell.time0-cell.time, p+k


def printKineticEnergy(cell):
    print '%f\t%f'% (cell.time0-cell.time, cell.kineticEnergy())


def printMomentum(cell):
    pX = pY = 0
    for a in cell.atoms:
        pX += a.xDot
        pY += a.yDot
    print >>cell.fh, '%f\t%f\t%f'% (cell.time0-cell.time, pX, pY)


def printLeftHalf(cell):
    global total, obs
    obs += 1
    num = 0
    for a in cell.atoms:
        if a.x <= cell.Lx/2:
            num += 1
```

```
    total += num

    print >>cell.fh, '%f\t%d\t%f'% (cell.time0-cell.time, num, total/obs)


def printAvgTemperature(cell):
    global Tsum, obs
    obs += 1
    Tsum += cell.kineticEnergy()/len(cell.atoms)
    print '%f\t%f'% (cell.time0-cell.time, Tsum/obs)


def printVirial(cell):
    global Vsum, obs
    Vol = 2*36   # dimension * volume
    obs += 1
    f = 0.146/8.62

    for a,b in cell.pairs():
        rx = abs(b.x - a.x)
        ry = abs(b.y - a.y)
        Vsum += abs(rx * a.force[0]) + abs(ry * a.force[1])

    print '%f\t%f'% (cell.time0-cell.time, f*Vsum/(obs*Vol))


def printPressure(cell):
    global Tsum, Vsum, obs
    V = 36
    obs += 1
    f = 0.146/8.62
    N = len(cell.atoms)

    Tsum += cell.kineticEnergy()/N
    for a,b in cell.pairs():
        rx = abs(b.x - a.x)
        ry = abs(b.y - a.y)
        Vsum += abs(rx * a.force[0]) + abs(ry * a.force[1])

    P = (N * (Tsum/obs) / V) + f*Vsum/(obs*2*V)
    print '%f\t%f'% (cell.time0-cell.time, P)


def printEnergy3(cell):
    #global data
    p = 0
    for a in cell.atoms:
        p += a.pot

    #data[cell.time0-cell.time] = p/len(cell.atoms)
    print '%f\t%f'% (cell.time0-cell.time, p/len(cell.atoms))


def printEnergy4(cell):
    # global data
    k = cell.kineticEnergy()
    p = 0
    for a in cell.atoms:
        p += a.pot
    tot = p + k
    temp = cell.kineticEnergy()/len(cell.atoms)
    press = cell.pressure()

    # data[cell.time0-cell.time] = p/len(cell.atoms)
    print '%f\t%f\t%f\t%f'% (cell.time0-cell.time, tot, temp, press)


def printEnergy5(cell):
    k = cell.kineticEnergy()
    p = 0
    for a in cell.atoms:
        p += a.pot
    tot = p + k
    temp = cell.kineticEnergy()/len(cell.atoms)
```

```
    press = cell.pressure()

    print '%f\t%f\t%f\t%f\t%f\t%f'% (cell.time0-cell.time, p, k, tot, temp, press)


def printEnergy6(cell):
    global E0, data

    k = cell.kineticEnergy()
    p = 0
    for a in cell.atoms:
        p += a.pot
    tot = p + k
    temp = cell.kineticEnergy()/len(cell.atoms)
    press = cell.pressure()
    if E0 == None:
        E0 = tot

    print '%f\t%f\t%f\t%f\t%f\t%f'% (cell.time0-cell.time, p, tot, tot-E0, temp, press)
    # print '%f\t%f\t%f\t%f\t%f\t%f'% (cell.time0-cell.time, p, tot, tot-E0, temp, press)


#############
##  MAIN  ##
#############

def usage():
    print '''
Usage:
  > python molec.py [OPTIONS]

'''
if __name__ == '__main__':

    if len(sys.argv) > 1:

        # help
        if sys.argv[1] == '-h':
            usage()


        # generate random configuration
        elif sys.argv[1] == '-g':

            # Set the values below as desired
             #cell = Cell({'size':6, 'cells':1, 'step':.0001, 'time':.1, 'nodisplay':True, 'menu':False,
'rk':False})
            #cell.randomAtoms(16)
                cell = load('8.5.a.initial', {'step':.0001, 'time':.1, 'nodisplay':True, 'menu':False,
'rk':False})

            #cell.update_extra = printEnergy
            #cell.fh = open('8.5.a.output', 'w')
            print 'No.CELLS:',len(cell.atoms)
            cell.animate()
            cell.save(sys.argv[2])
            #cell.fh.close()


        # Read from file
        elif sys.argv[1] == '-f':
            totalP = totalK = obs = 0
            cell = load(sys.argv[2], {'nodisplay':True, 'menu':False})
            cell.update_extra = printEnergy
            cell.fh = open('8.3.a.output', 'w')
            cell.animate()
            cell.fh.close()


        # 8.2.a
        elif sys.argv[1] == '8.2.a':
            totalP = totalK = obs = 0
              #cell = Cell({'size':6, 'sizeY':6, 'cells':1, 'step':.0001, 'time':.1, 'nodisplay':False,
'menu':False})
```

```
             cell = Cell({'size':3, 'sizeY':3, 'cells':40, 'step':.00001, 'time':.1, 'nodisplay':False,
'menu':False})
            cell.update_extra = printEnergy
            cell.fh = open('8.2.a.2.output', 'w')
            cell.randomAtoms(16)
            cell.animate()
            cell.fh.close()

        # 8.2.b
        elif sys.argv[1] == '8.2.b':
            totalP = totalK = obs = 0
               cell = Cell({'size':12, 'sizeY':6, 'cells':1, 'step':.0001, 'time':2, 'nodisplay':True,
'menu':False})
            cell.update_extra = printEnergy
            cell.fh = open('8.2.b.output', 'w')
            cell.randomAtoms(16)
            cell.animate()
            cell.fh.close()

        # 8.2.c
        elif sys.argv[1] == '8.2.c':
            totalP = obs = 0
            cell = Cell({'size':12, 'sizeY':6, 'cells':1, 'step':.0001, 'time':.2, 'nodisplay':True})
            cell.update_extra = printLeftHalf
            cell.fh = open('8.2.c.output', 'w')
            cell.randomAtoms(64)
            print 'No.CELLS:',len(cell.atoms)
            cell.animate()

        # 8.4.a
        elif sys.argv[1] == '8.4.a':
            totalP = totalK = obs = e0 = e1 = 0
            cell = load('8.4.a.initial', {'nodisplay':True, 'menu':False})
            cell.step = float(sys.argv[2])
            cell.update_extra = printEnergy
            cell.fh = open('8.4.a.'+str(cell.step)+'.output', 'w')
            cell.animate()

        # 8.4.b
        elif sys.argv[1] == '8.4.b':
            cell = load('8.4.b.initial', {'nodisplay':True, 'menu':False})
            cell.update_extra = printMomentum
            cell.fh = open('8.4.b.output', 'w')
            cell.animate()

        # 8.4.c
        elif sys.argv[1] == '8.4.c':
            data = {}
            cell = load('8.4.c.initial', {'time':.1, 'nodisplay':True, 'menu':False})
            cell.update_extra = printEnergy2
               for t in [.0001, .000095, .00009, .000085, .00008, .000075, .00007, .000065, .00006, .
000055, .00005, .000045, .00004, .000035, .00003, .000025, .00002, .000015, .00001, .000005]:
                cell.step = t
                cell.animate()
                d = Data(data)
                d.minimize()
                print '%f\t%f\t%f'% (t, d.line[0], d.dev)

        # 8.4.d
        elif sys.argv[1] == '8.4.d':
            data = {}
              cell = load('8.4.d.3.initial', {'step':.001, 'time':.1, 'nodisplay':False, 'menu':False,
'rk':True})
            cell.update_extra = printEnergy2
              # for t in [.0001, .000095, .00009, .000085, .00008, .000075, .00007, .000065, .00006, .
000055, .00005, .000045, .00004, .000035, .00003, .000025, .00002, .000015, .00001, .000005]:
            for t in [.001]:
                cell.step = t
                # cell.randomAtoms(30)
                cell.animate()
                d = Data(data)
                d.minimize()
                d.plot(); exit()
                d.print_slope_dev(t)
```

```
        # 8.5.a
        elif sys.argv[1] == '8.5.a':
            # data = {}
            Vsum = Tsum = obs = 0
                cell = load('8.5.a.initial', {'step':.0001, 'time':10, 'nodisplay':True, 'menu':False,
'rk':False})
            # cell.update_extra = printAvgTemperature
            # cell.update_extra = printEnergy2
            # cell.update_extra = printKineticEnergy
            # cell.update_extra = printVirial
            cell.update_extra = printPressure
            cell.animate()
            # d = Data(data)
            # d.minimize()
            # d.plot()

        # 8.5.c
        elif sys.argv[1] == '8.5.c':
            # data = {}
            Vsum = Tsum = obs = 0
                cell = load('8.5.a.initial', {'rescale':2, 'step':.0001, 'time':10, 'nodisplay':True,
'menu':False, 'rk':False})
            # cell.update_extra = printPressure
            cell.update_extra = printAvgTemperature
            cell.animate()
            # d = Data(data)
            # d.minimize()
            # d.plot()

        # 8.8.a
        elif sys.argv[1] == '8.8.a':
            data = {}
            cell = Cell({'size':6, 'cells':3, 'step':.0001, 'time':1, 'nodisplay':True, 'menu':False})
            cell.setup(int(sys.argv[2]), 't')
            # print 'Num atoms:', len(cell.atoms)
            cell.update_extra = printEnergy3
            # cell.update_extra = printPressure
            cell.animate()
            d = Data(data)
            d.minimize()
            d.plot()

        # 8.8.b
        elif sys.argv[1] == '8.8.b':
            data = {}
            cell = Cell({'size':6, 'cells':1, 'step':.00001, 'time':1, 'nodisplay':True, 'menu':False})
            cell.setup(int(sys.argv[2]), 's')
            # print 'Num atoms:', len(cell.atoms)
            cell.update_extra = printEnergy3
            # cell.update_extra = printPressure
            cell.animate()
            #d = Data(data)
            #d.minimize()
            #d.printout()
            #d.plot()

        # 8.9.a
        elif sys.argv[1] == '8.9.a':
            data = {}
            cell = Cell({'size':6, 'cells':3, 'step':.0001, 'time':1, 'nodisplay':True, 'menu':False})
            cell.setup(16, 't')
            #print 'Num atoms:', len(cell.atoms)
            #cell.update_extra = printEnergy4
            # cell.update_extra = printPressure
            cell.animate()
            cell.save(sys.argv[2])

        # 8.9.b
        elif sys.argv[1] == '8.9.b':
            cell = load('8.9.b.initial', {'rescaleV':4, 'step':.0001, 'nodisplay':True, 'menu':False})
            cell.update_extra = printEnergy5
            cell.animate()
```

```
        # 8.9.d
        elif sys.argv[1] == '8.9.d':
            scale = 3
            E0 = None
            data = {}
                    cell  =  load('8.9.d.initial',  {'rescaleV':scale,  'step':.0001,  'nodisplay':False,
'menu':False})
            cell.update_extra = printEnergy6
            cell.animate()

        # 8.9.e
        elif sys.argv[1] == '8.9.e':
            cell = load('8.9.e.initial', {'cells':2, 'step':.0001, 'nodisplay':False, 'menu':False})
            cell.update_extra = printEnergy5
            cell.animate()


    # Default
    else:
        cell = Cell({'size':6, 'cells':1, 'step':.0001, 'time':10})
        cell.menu()
```

Supporting code:

```
from __future__ import with_statement, division, absolute_import

import sys, os, re, pygame, math
import pygame
from pygame.locals import *
from random import randint
from util import lv


global stdSize
stdSize = 768


class Animation:

    # constructor
    def __init__(self, vSizeX=10, vSizeY=10, cells=1, options={}):
        if options.has_key('nodisplay'):
            self.nodisplay = options['nodisplay']
        else:
            self.nodisplay = False

        if vSizeX < 1:
            vSizeX = 1
        if vSizeY < 1:
            vSizeY = 1
        global stdSize
        if vSizeX > vSizeY:
            self.size = (stdSize, int(stdSize*(vSizeY/vSizeX)))
            self.scale = (stdSize/vSizeX)/cells
        elif vSizeX < vSizeY:
            self.size = (int(stdSize*(vSizeX/vSizeY)), stdSize)
            self.scale = (stdSize/vSizeX)/cells
        else:
            self.size = (stdSize, stdSize)
            self.scale = (stdSize/vSizeX)/cells

        self.cells = cells
        if not self.nodisplay:
            pygame.init()
            # self.screen = pygame.display.set_mode(size, pygame.FULLSCREEN)
            self.screen = pygame.display.set_mode(self.size)
```

```
        self.clear()
    else:
        self.screen = None

# quit
def quit(self):
    pygame.quit()

# clear screen
def clear(self):
    self.screen.fill([255,255,255])

# check for events
def checkEvent(self):
    if self.nodisplay:
        return
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()
        if event.type == KEYDOWN:
            key = int(event.key)
            if ( key == 103
                 or key == 106
                 or key == 113 ): pass
            elif event.key == K_ESCAPE:
                self.clear()
                exit()
            # else:
            #    return int(event.key)

# set caption
def setcaption(self, cap):
    pygame.display.set_caption(cap)

# maintain
def maintain(self):
    while True:
        self.checkEvent()

# draw
def draw(self):
    if not self.nodisplay:
        pygame.display.flip()



# class Shape
class Shape:
    colorname = {}
    colorname['black'] = 0, 0, 0
    colorname['white'] = 255, 255, 255
    colorname['dkgrey'] = 80, 80, 80
    colorname['grey'] = 180, 180, 180
    colorname['random'] = randint(50,255), randint(50,255), randint(50,255)


# Circle
class Circle(Shape):

    # constructor
    def __init__(self, state, scale, size, cells=1, options={}):
        global stdSize

        self.state = state
        if options.has_key('nodisplay'):
            self.nodisplay = options['nodisplay']
            if self.nodisplay:
                return
        else:
            self.nodisplay = False

        self.scale = scale
        self.size = size
        self.cells = cells
        self.width = size[0]/cells
```

```
            self.height = size[1]/cells
            self.state['radius'] = int(scale* self.state['radius'])

            if type(self.state['color']) == type(''):
                self.state['color'] = self.colorname[self.state['color']]

            x,y = self.state['position']
            x = int(self.scale* x)
            y = int(self.scale* y)

            for i in range(self.cells):
                for j in range(self.cells):
                    pygame.draw.circle( self.state['screen'],
                                        self.state['color'],
                                        [int(x+i*self.width), int(y+j*self.height)],
                                        self.state['radius'] )

        # update
        def update(self, newstate):
            x,y = self.state['position']
            x = int(self.scale* x)
            y = int(self.scale* y)

            if not self.nodisplay:
                for i in range(self.cells):
                    for j in range(self.cells):
                        try:
                            pygame.draw.circle( self.state['screen'],
                                                self.colorname['white'],
                                                [int(x+i*self.width), int(y+j*self.height)],
                                                self.state['radius'] )
                        except Exception, e:
                            print "ERROR 1 in Circle.update:",e
                            exit()

            for k in newstate.iterkeys():
                self.state[k] = newstate[k]

            if self.nodisplay:
                return

            x,y = self.state['position']
            x = int(self.scale* x)
            y = int(self.scale* y)

            for i in range(self.cells):
                for j in range(self.cells):
                    try:
                        #print int(x+i*self.width),',', int(y+j*self.height)
                        pygame.draw.circle( self.state['screen'],
                                            self.state['color'],
                                            [int(x+i*self.width), int(y+j*self.height)],
                                            self.state['radius'] )
                    except Exception, e:
                        print "ERROR 2 in Circle.update:",e
                        exit()


    def mkCircle(self, position, radius=1, color=[80,80,80]):
        state = ({ 'screen':self.screen,
                   'color':color,
                   'position':position,
                   'radius':radius })
        cir = self.Circle(state, self.scale, self.size, self.cells, {'nodisplay':self.nodisplay})
        return cir



# add elements of two lists component by component
#     even lists of different sizes
def sumLists(lists):
    out = []
    i = 0
```

```
    active = True
    while active == True:
        active = False
        for list in lists:
            if len(list) >= i+1:
                if len(out) >= i+1:
                    out[i] += list[i]
                else:
                    out.append(list[i])
                active = True
        i += 1
    return out

# Multiply each element of a list by a factor
def multList(x, list):
    out = []
    for e in list:
        out.append(x*e)
    return out


def simplify_basic(line):

    line = line.replace( '\\', ' ' )
    line = line.replace( '/', ' ' )
    line = line.replace( '-', ' ' )
    line = re.sub( '\s+', ' ', line )
    line = re.sub( '^\s+', '', line )
    line = re.sub( '\s+$', '', line )

    return line


##  Default settings for graphics output
chart_object.set_defaults(line_plot.T, line_style=None)
theme.scale_factor = 10
theme.output_format = 'png'
theme.use_color = True
theme.reinitialize()
fill_styles = [fill_style.red,
               fill_style.red,
               fill_style.black,
               fill_style.blue,
               fill_style.darkseagreen,
               fill_style.aquamarine1,
               fill_style.gray70,
               fill_style.brown,
               fill_style.green,
               fill_style.darkorchid,
               fill_style.yellow,
               fill_style.goldenrod,
               fill_style.white]


##  Plot class definition
class Plot:
    '''
    Create graph object to easily plot data

    data :  [ [x1, y1], [x2, y2], [x3, y3], .... ]
    options :  { 'text': words at top
                 'xcol': list of columns in data vector for x-coord
                 'ycol': list of columns in data vector for y-coord
                 'lineLabel': list of labels for y-columns
                 'xRange': [low, high]
                 'yRange': [low, high]
                 'xTics':  num of tics
                 'yTics':  num of tics
                 'xFormat': num format
                 'yFormat': num format
                 'line_style': line style
    '''
    # Constructor
    def __init__(self, data, options={}):
```

```
    self.data = data

    if options.has_key('xcol'):
        self.xcol = options['xcol']
    else:
        self.xcol = [0]
    if options.has_key('ycol'):
        self.ycol = options['ycol']
    else:
        self.ycol = range(len(data[0]))[1:]
    if options.has_key('title'):
        self.title = options['title']
    else:
        self.title = ''
    if options.has_key('fill_style'):
        self.fill_style = options['fill_style']
    else:
        self.fill_style = fill_styles
    if options.has_key('xLabel'):
        self.xLabel = options['xLabel']
    else:
        self.xLabel = ''
    if options.has_key('yLabel'):
        self.yLabel = options['yLabel']
    else:
        self.yLabel = ''
    if options.has_key('lineLabel'):
        self.lineLabel = options['lineLabel']
    else:
        self.lineLabel = 'line'
    if options.has_key('xRange'):
        self.xRange = options['xRange']
    else:
        self.set_xRange()
    if options.has_key('yRange'):
        self.yRange = options['yRange']
    else:
        self.set_yRange()
    if options.has_key('xTics'):
        self.xTics = options['xTics']
    else:
        self.xTics = 5
    if options.has_key('yTics'):
        self.yTics = options['yTics']
    else:
        self.yTics = 5
    if options.has_key('xFormat'):
        self.xFormat = "/a-60/hL%" + options['xFormat']
    else:
        xR = self.xRange[1] - self.xRange[0]
        if xR > .001 and xR < 10000:
            self.xFormat = "/a-60/hL%0.4f"
        else:
            self.xFormat = "/a-60/hL%0.5e"
    if options.has_key('yFormat'):
        self.yFormat = "/a-60/hL%" + options['yFormat']
    else:
        yR = self.yRange[1] - self.yRange[0]
        if yR > .001 and yR < 10000:
            self.yFormat = "%0.4f"
        else:
            self.yFormat = "%0.5e"
    if options.has_key('lineStyle'):
        self.lineStyle = options['lineStyle']
    else:
        self.lineStyle = None
    self.xTic = abs( (self.xRange[1] - self.xRange[0])/self.xTics )
    self.yTic = abs( (self.yRange[1] - self.yRange[0])/self.yTics )


# set xRange automatically from data
def set_xRange(self):
    low = sys.maxint
    high = -sys.maxint
```

```
        for x in self.xcol:
            for dat in self.data:
                if dat[x] > high:
                    high = dat[x]
                if dat[x] < low:
                    low = dat[x]
        self.xRange = [low, high]


    # set yRange automatically from data
    def set_yRange(self):
        low = sys.maxint
        high = -sys.maxint
        for y in self.ycol:
            for dat in self.data:
                if dat[y] > high:
                    high = dat[y]
                if dat[y] < low:
                    low = dat[y]
        self.yRange = [low, high]


    # printout (should pipe into a file)
    def printout(self):
        titleLocation = (30,120)
        if self.xRange[0] == self.xRange[1]:
            self.xRange = [self.xRange[0]-1, self.xRange[0]+1]
            self.xTic = .5
        if self.yRange[0] == self.yRange[1]:
            self.yRange = [self.yRange[0]-1, self.yRange[0]+1]
            self.yTic = .5

        xAxis = axis.X(format=self.xFormat, tic_interval=self.xTic, label=self.xLabel)
        yAxis = axis.Y(format=self.yFormat, tic_interval=self.yTic, label=self.yLabel)
        tb = text_box.T(loc=titleLocation, text=self.title, line_style=self.lineStyle)
        ar = area.T(x_axis=xAxis, y_axis=yAxis, x_range=self.xRange, y_range=self.yRange)

        # Iterate over appropriate columns to make plots
        plots = []
        for x in self.xcol:
            j = -1
            for y in self.ycol:
                j += 1
                lineLabel = 'line'
                try:
                    if type(self.lineLabel) == type([]):
                        lineLabel=self.lineLabel[j]
                except:
                    print 'ERROR in printout()'
                tick = tick_mark.Circle(size=2, fill_style=self.fill_style[y], line_style=None)
                plot = line_plot.T(label=lineLabel, data=self.data, xcol=x, ycol=y, tick_mark=tick)
                ar.add_plot(plot)

        ar.draw()
        tb.draw()



# Plot data (should pipe into png file)
def plot_png(data, options={}):
    '''
    plot_png will plot a set of data, label/format it, and save at as a png to <file>

    data :  [ [x1, y1], [x2, y2], [x3, y3], .... ]
    options :  { 'text': words at top
                 'lineLabel': list of labels for y-columns
                 'xcol': list of columns in data vector for x-coord
                 'ycol': list of column in data vector for y-coord
                 'xRange': [low, high]
                 'yRange': [low, high]
                 'xTics':  num of tics
                 'yTics':  num of tics
                 'xFormat': num format
                 'yFormat': num format
```

```
                'line_style': line style
    '''
    plot = Plot(data, options)
    plot.printout()


# Read from a file and plot data
def plotFromFile(file, options={}):

    f = open(file)
    data = []
    for line in f:
        line = line.strip()
        dat = line.split("\t")
        dataLine = []
        for d in dat:
            dataLine.append(float(d))
        data.append(dataLine)

    labels = []
    i = 0
    for d in data[0]:
        labels.append('col '+str(i))
        i += 1

    options['title'] = simplify_basic(file)
    if not options.has_key('lineLabel'):
        options['lineLabel'] = labels

    plot_png(data, options)



##  Data CLASS  ##

class Data:

    def __init__(self, data, degree=1):
        ''' <data> is a dictionary s.t.: data[x] = y '''
        self.data = data
        self.degree = degree
        self.X = sorted(data.keys())
        self.Y = sorted(data.values())
        self.medX = average(self.X)
        self.medY = average(self.Y)
        self.setCurve(degree)
        self.dev = self.getStdDev(self.error)


    def setCurve(self, degree):
        if degree == 1:
            self.setLine()


    def avgSlope(self):
        n = -1
        thetas = []
        lastX = self.X[0]
        for x in self.X[1:]:
            # if x == self.medX:
            #    continue
            n += 1
            # rise = self.data[x] - self.medY
            # run = x - self.medX
            rise = self.data[x] - self.data[lastX]
            run = x - lastX
            thetas.append( math.atan2(rise, run) )
            lastX = x
        theta = average(thetas)
        slope = math.tan(theta)
        return slope


    def setLine(self):
```

```
    last = self.X[0]
    m = self.avgSlope()
    x = self.medX
    y = self.medY
    b = y - m*x
    self.line = [m,b]

    def curve(z):
        return m*z + b
    self.curve = curve

    def error(z):
        return (self.data[z] - m*z - b)**2
    self.error = error


def getStdDev(self, error):
    e = 0
    for x in self.X:
        e += error(x)
    return math.sqrt(e)/(len(self.X))


def best(self, i):
    if self.degree == 1:
        line = self.line
        x = line[i]
        best = x
        g = .5*x
        lastDev = self.dev
        dev = sys.maxint
        while abs(g) > 1e-10:
            x += g
            line[i] = x
            m,b = line
            def error(z):
                return (self.data[z] - m*z - b)**2
            dev = self.getStdDev(error)
            # lv({'m':x, 'dev':dev, 'g':g, 'best':[best,self.dev]})

            # Better value found
            if dev <= self.dev:
                best = x
                self.dev = dev

            if dev > lastDev:
                g = -.9*g

            lastDev = dev


    return best


def minimize(self):
    if self.degree == 1:
        m,b = self.line
        m = self.best(0)    # pass the index in self.line
        b = self.best(1)
        m = self.best(0)
        b = self.best(1)
        m = self.best(0)
        b = self.best(1)
        self.line = [m,b]

        def curve(z):
            return m*z + b
        self.curve = curve

        def error(z):
            return (self.data[z] - m*z - b)**2
        self.error = error
```

```
    def plot(self, options={}):
        points = []
        for x in self.X:
            points.append([x, self.data[x], self.curve(x)])

        subj = options['lineLabel']
        options['lineLabel'] = [subj, 'Drift:'+str(self.line[0])]
        if not options.has_key('ycol'):
            options['ycol'] = [1,2]
        if not options.has_key('title'):
            options['title'] = '%s: %f+-%0.3e'% (subj, self.medY, self.dev)

        plot_png(points, options)


    def printout(self):
        for x in self.X:
            print '%f\t%f\t%f'% (x, self.data[x], self.curve(x))


    def print_slope_dev(self, t):
        print '\t%f\t\t%0.4e\t\t%0.4e'% (t, self.line[0], self.dev)



def average(list):
    sum = 0
    for e in list:
        sum += e
    return sum/len(list)
```