Physics 2300
Spring 2018

Name _____

Lab partner _____

Project 5: Molecular Dynamics

If a computer can model three mutually interacting objects, why not model more than three? As you'll soon see, there is little additional difficulty in coding a simulation of arbitrarily many interacting particles. Furthermore, today's personal computers are fast enough to simulate the motion of hundreds of particles "while you wait," and to animate this motion at reasonable frame rates while the calculations are being performed.

The main difficulty in designing a many-particle simulation is not in the simulation itself but rather in deciding what we want to learn from it. Predicting the individual trajectories of a hundred particles is usually impractical because these trajectories are chaotic. Even if we could make such predictions, the sheer amount of data would leave us bewildered. Instead, we'll want to focus on higher-level patterns and statistical data.

In this project we'll also shift our attention from the very large to the very small—from planets to molecules. The main goal will be to learn how the familiar properties of matter arise from motions at the molecular scale.

Molecular forces

Under ordinary circumstances, molecules are electrically neutral. This implies that the forces between them are negligible when they're far apart. However, when two molecules approach each other, the distortion of their electron clouds usually produces a weak attractive force. When they get too close, the force becomes strongly repulsive (see Figure 1). For all but the simplest molecules, these forces also depend on the molecules' shapes and orientations. In this project we'll ignore such complications and just simulate the behavior of spherically symmetric molecules such as noble gas atoms.

Figure 1: When two molecules are near each other, there is a weak attractive force between them. When they get too close, the force becomes strongly repulsive.

Even for the simplest molecules, there is no simple, exact formula for the intermolecular force. Fortunately, we don't really need an exact formula; any approximate formula with the right general behavior will give us useful results. The formula that is most commonly used to model the interaction between noble gas atoms is the *Lennard-Jones potential*,

$$U(r) = 4\epsilon\left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6}\right]. \tag{1}$$

This is a formula for the potential energy, $U(r)$, in terms of the distance r between the centers of the two interacting molecules. The constants r_0 and ϵ represent the approximate molecular diameter and the overall strength of the interaction. The numerical values of these constants will depend on the specific type of molecule; the table below gives some values obtained by fitting the Lennard-Jones model to experimental data for noble gases at low densities.

	r_0 (Å)	ϵ (eV)
helium	2.65	0.00057
neon	2.76	0.00315
argon	3.44	0.0105

Figure 2 shows a graph of the Lennard-Jones potential. When $r \gg r_0$, the energy is negative and approaches zero in proportion to $1/r^6$. This is the correct behavior of the so-called van der Waals force, and can be derived from quantum

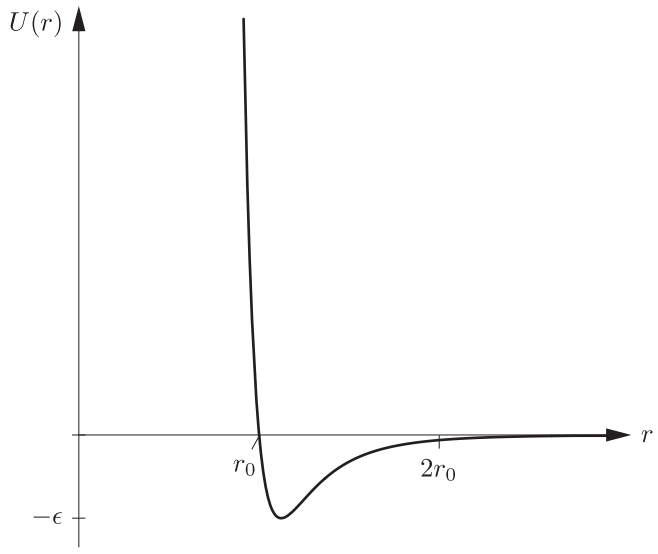

Figure 2: The Lennard-Jones potential energy function (equation 1).

theory. When $r < r_0$ the energy becomes positive, rising very rapidly to give a strong repulsive force as r decreases. John Lennard-Jones suggested using a power law to model this repulsive behavior, and nowadays we normally take the power to be -12 for computational convenience. We'll stick with this choice because no simple improvement to it would give significantly more accurate results.

Exercise: Find the value of r (in terms of r_0) at which the Lennard-Jones function reaches its minimum, and show that its value at that point is $-\epsilon$.

Exercise: Consider two molecules, located at (x_1, y_1) and (x_2, y_2), interacting via the Lennard-Jones force. Find formulas for the x and y components of the force acting on each molecule, in terms of their coordinates and their separation distance r. (Hint: First calculate the r component of the force, which is given by the general formula $F_r = -dU/dr$. Then draw a picture, and be careful with minus signs.)

Units

Once again we can make our lives easier by choosing units that are natural to the system being modeled. For a collection of identical molecules interacting via the Lennard-Jones force, a natural system of units would set r_0, ϵ, and the molecular mass (m) all equal to 1. These three constants then become our units of distance, energy, and mass. Units for other mechanical quantities such as time and velocity are implicitly defined in terms of these.

Exercise: What combination of the constants r_0, ϵ, and m has units of time?

Exercise: If we are modeling argon atoms using natural units, what is the duration of one natural unit of time, expressed in seconds? (Use the values of r_0 and ϵ from the table above. Note that the r_0 values are given in Ångström units (Å), where 1 Å$= 10^{-10}$ m, while the ϵ values are given in electron-volts (eV), where 1 eV $= 1.60 \times 10^{-19}$ J.)

Exercise: Suppose that an argon atom has a speed of 1, expressed in natural units. What is its speed in meters per second?

Another quantity that we'll eventually want to determine for our collection of molecules is temperature. To define a natural unit of temperature we can set Boltzmann's constant k_B (which is essentially a conversion factor between molecular energy and temperature) equal to 1. In conventional units,

$$k_B = 1.38 \times 10^{-23} \text{ J/K} = 8.62 \times 10^{-5} \text{ eV/K}. \tag{2}$$

I'll explain later how to actually determine the temperature of a system from the speeds of the particles.

Exercise: Suppose that a collection of argon atoms has a temperature of 1 in natural units. What is its temperature in kelvin? (For comparison, the boiling point of argon at atmospheric pressure is 87 K.)

Exercise: Repeat the previous exercise for helium, and discuss the result briefly.

Lists

To simulate the motion of just *two* noble gas atoms, you could simply modify your `Orbit2` program to use the Lennard-Jones force law instead of Newton's law of gravity. Recall that in that program you used the variables `x1`, `y1`, `vx1`, and so on for the first planet, and `x2`, `y2`, `vx2`, and so on for the second planet. As you can imagine, this approach becomes awkward if you add more planets (or atoms). Fortunately, Python (like nearly all programming languages) provides a convenient mechanism for working with a collection of related variables: a *list* (also sometimes called an *array*), which you refer to by a single variable name such as `x` or `vx`.

The elements of a Python list are numbered from 0 up to some maximum value. To access a particular element, you put that element's number in square brackets after the name of the list:

```
x[14] = 4.2
fib[2] = fib[0] + fib[1]
```

The number in brackets is called an *index*. Because the index of a list's first element is zero, the index of the last element is always one less than the size of the list. For example, if `x` is a list of 100 numbers, then you access those numbers by typing `x[0]`, `x[1]`, and so on up to `x[99]`. (This convention is common to Python and all the C-derived languages, including Java and JavaScript. But there are other languages, such as Fortran and Mathematica, in which list indices start at 1 rather than 0.)

This bracket-index notation would be no improvement at all if the quantity in brackets always had to be a literal number. But it doesn't: You're allowed to put

any integer-valued variable, or any integer-valued expression, inside the brackets. So, for instance, if `x` is a list of 100 elements, you could add up all the elements with this `while` loop:

```
sum = 0
i = 0
while i < 100:
    sum += x[i]
    i += 1
```

Notice that the condition on the loop is a strict `<`, not `<=`, because `x[100]` doesn't exist.

Although there's nothing wrong with the preceding `while` loop, Python (like most programming languages) provides a more compact way of executing a block of code a fixed number of times: a `for` loop. To add up the elements of the `x` list with a `for` loop, you would type the following:

```
sum = 0
for i in range(100):
    sum += x[i]
```

This code also uses Python's `range` function, which in this case effectively produces a list of the integers 0 through 99 (not including 100!). Both the `range` function and the `for` loop can be used in other ways, and are a bit difficult to explain in general, but for looping over the elements of a list, this example is all you need to know. (If you've used `for` loops in other languages, you may need to revise some of your expectations about how you can and cannot use a `for` loop in Python. I prefer to use `for` to loop over list elements, and in other situations where the number of iterations is explicitly known in advance; for any other looping situation I use `while`.)

And how do you create a Python list in the first place? If the list is sufficiently short, you can create it by providing literal values, in brackets, separated by commas:

```
daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Often, though, we want to initialize a list in a more automated way, and awkwardly, the best method in that case depends on what Python environment you're using. The most robust method, I think, is to first create an empty list and then write a loop to add elements to it using the list's `append` function:

```
x = []
for i in range(100):
    x.append(0)
```

In this example I've given every list element an initial value of 0, but you could pass any other initial value (possibly depending on `i`) to the `append` function.

The program design

You're nearly ready to start writing a molecular dynamics simulation program. But there are several decisions to make before you can actually start coding, and to save time I've made some of these decisions for you.

First decision: You'll simulate a collection of noble gas atoms in *two* dimensions, not three. A 3D simulation would be only a tiny bit harder to code in GlowScript-VPython, and you can certainly try it later if you like, but in 3D it's hard to see what's happening, because the atoms in front tend to block your view of the ones behind. Fortunately, there's plenty to be learned from a 2D molecular dynamics simulation.

Second decision: The atoms will live in a square "box" whose walls are located at the edges of the graphics scene. So you should set `scene.width` and `scene.height` to be the same; you can decide exactly how many pixels they should be, depending on your screen size. The x and y coordinates inside the box will each run from 0 up to a maximum value that we'll call `w` (for *width*), measured in natural units as defined above (multiples of r_0). You'll set `w = 20` initially, but plan on changing the value later. Then you should set `scene.center` to be at the middle of the box, and set `scene.range` to put the edges at 0 and `w`. Set `scene.fov` to a small value like 0.01, to eliminate distortion as in the previous project. Disable auto-scaling, zooming, and rotation.

Third decision: Use the variable name `N` for the total number of atoms; use `x`, `y`, `vx`, `vy`, `ax`, and `ay` for the lists that will hold the atoms' positions, velocities, and accelerations; and use a list called `ball` for the `sphere` objects that will represent the atoms in the graphics scene. Set `N = 100` for now, but write your code to work for any reasonable value of `N`.

Exercise: Create a new GlowScript program called `MD`, and put code in it to implement the design decisions described above. Use a `for` loop to initialize all the velocities and accelerations to zero, to initialize the positions to a common location near the middle of the box, and to initialize each element of the `ball` list to a `sphere` at the corresponding position, with diameter 1.0, in your favorite color. Run the program and check that you see a single sphere at the correct position, with the correct size. (You'll see only one sphere, because they're all at the same position.)

Exercise: Modify your initialization code to place all the atoms at different positions inside the box, so none of the spheres that represent them overlap. The easiest way to do this is to arrange them in regular rows, but this is still a bit tricky because you need to start a new row whenever the old row is full. You'll need a couple of variables to keep track of the x and y locations where the next atom goes (or where the last one went). Spend some time on this task, trying out your ideas to see if they work, but if you and your lab partner can't come up with a working algorithm within 15 minutes or so, be sure to ask someone else for a hint. Once you get your code working for `N = 100` and `w = 20`, change these values and make sure

it still works—but don't worry about what happens when the box is too small to comfortably hold all the atoms.

Coding the simulation

Now you can start adding code to put the atoms into motion.

Exercise: Create an infinite `while` loop for your main simulation loop. Include a `rate` function with a parameter of 1000 or more, so the loop will run as fast as possible. During each iteration this loop should call a function called `singleStep` about 10 times; this is the number of calculation steps per animation frame (and you can fine-tune it later). After all 10 calls to `singleStep`, update the positions of all the `ball` objects using the current values of `x` and `y`. In the `singleStep` function itself, put in some temporary code that merely changes `x[0]` and `y[0]` by a tiny amount. Test your program to verify that the first atom-sphere moves as expected.

Exercise: Now remove the temporary code from `singleStep` and replace it with code to implement the Verlet algorithm: First update all the positions and update the velocities half-way, then call a separate function (call it `computeAccelerations`) to compute all the new accelerations, and finally update the velocities the remaining half-way. Use a fixed time step of 0.02 in natural units. Create the `computeAccelerations` function, but for now, just put some temporary code in it that sets `ax[0]` and `ay[0]` to some tiny nonzero values. Again, test your program to make sure it behaves as expected.

The `computeAccelerations` function must have two parts: one to calculate the accelerations from collisions with the walls of the box, and one to calculate the accelerations from collisions between atoms. The collisions with the walls are potentially trickier, especially if we want infinitely stiff walls that produce instantaneous changes in velocity. Instead, it's easier to make the walls "soft," so the force they apply increases gradually as the edge of an atom moves farther into the wall. A linear "spring" force works well; here is a code fragment that implements such a force for the vertical walls:

```
for i in range(N):
    if x[i] < 0.5:
        ax[i] = wallStiffness * (0.5 - x[i])
    elif x[i] > (w - 0.5):
        ax[i] = wallStiffness * (w - 0.5 - x[i])
    else:
        ax[i] = 0.0
```

The Python word `elif` is short for "else if"; I hope this example of its use is self-explanatory. Here `wallStiffness` is the "spring constant," for which a value of 50 in natural units works pretty well. Notice that since $m = 1$, the force on a molecule is the same as its acceleration.

Question: Why does the `if` statement in this code test whether `x[i]` is less than 0.5, rather than testing whether `x[i]` is less than 0?

Exercise: Insert this code, and similar code to handle collisions with the horizontal walls, into your `computeAccelerations` function. To test your code, give atom 0 a nonzero velocity along some diagonal direction. You should then see this atom bounce around inside the box.

Exercise: Now add the code to handle collisions between atoms. You'll need a double loop over all pairs of atoms:

```
for i in range(N):
    for j in range(i):
        # compute forces between atoms i and j
        # and add on to the accelerations of both
```

To compute the force components, use the formulas you worked out at the beginning of this project. Be careful with minus signs. Then run your code and enjoy the show!

Question: Why does the inner loop (over `j`) run only from 0 up to `i-1`, instead of all the way up to `N-1`?

Optimizing performance

Ninety percent of the time, you shouldn't worry about writing code that will run as fast as possible. It's much more important to make your code simple and easy for a *human* to read and understand. *Your time is more valuable than the computer's!*

However, your `singleStep` and `computeAccelerations` functions fall in the other ten percent. These functions contain loops that execute the same code thousands upon thousands of times. Any effort that you spend speeding up the code inside these loops will be rewarded in proportion.

Exercise: A typical atom in this simulation might have a speed of 1 in natural units. How many units of time will it take this atom to cross from one side of the box to the other (assuming no collisions along the way)? How many calls to `singleStep` are required for such a one-way trip? During this same time, how many times will the code within each of the loops in `singleStep` be executed? How many times will the code within the double loop in `computeAccelerations` be

executed? How many times will your code change the position of a `sphere` graphics object? (Please assume that `N = 100`, `w = 20`, and there are 10 calculation steps per animation frame—even if you've changed these variables in your program. Show your calculations and answers clearly in the space below.)

Here, then, are some tips for optimizing the performance of your MD simulation:

- Don't spend time trying to optimize performance until after you're sure that your program is working correctly.
- Never worry about optimizing code that isn't executed at least thousands of times per second.
- Addition, subtraction, multiplication, and division are fast, but exponentiation (`**`) and function calls (such as `sqrt`) are slow. Therefore, when you compute the Lennard-Jones force, avoid all uses of exponentiation and of function calls. Since only even powers of r appear in the force, you can work with r^2 (or better, $1/r^2$) instead of r. To square or cube a number, just multiply it by itself. Cube $1/r^2$ to get $1/r^6$, then square that to get $1/r^{12}$. Use intermediate variables where necessary to avoid repeating calculations.
- To speed up the force calculations even more, don't bother to compute the force between two atoms when they're farther than (say) 3 units apart. (Be sure to test whether they're too far apart in a way that avoids calculating a square root.)
- The `singleStep` function makes repeated use of the combinations $dt/2$ and $dt^2/2$, where dt is the time step. So compute these quantities once and for all in your program's initialization section, storing them in global variables.
- A brute-force way to speed up the simulation is to increase dt. Naturally, this will also increase the truncation error. Don't try this until later, when you'll have a way of checking whether energy is approximately conserved.

- Graphics can often be a performance bottleneck. In GlowScript, there is overhead associated with changing any of the attributes of a `sphere` (or any other graphics object). That's why I've told you to use the separate variables `x` and `y` for the physics calculations, updating the `ball.pos` values only once per animation frame. On the other hand, you still want the animation to be reasonably smooth if possible, and typically this requires at least 20 or 30 animation frames per second. Try adjusting the number of calculation steps per animation frame, and see what value seems to give the best results.
- If you want to monitor performance quantitatively, VPython provides a `clock` function that returns the current time in seconds. Call it twice and subtract the values to determine how much time passed in between calls. (Measuring the performance in this way is *optional* for this project.)

Exercise: Spend some time optimizing the performance of your program, and describe the effects in the space below. Which changes seem to make the most difference? After optimizing, how large can you make `N` and still get reasonably smooth animation?

GUI controls

Your program still lacks two important features: It doesn't let you control the simulation in any way while it is running, and it doesn't give you any quantitative data to describe what's going on. You could add all sorts of features of both types. Feel free to go beyond what the following instructions require!

Exercise: Add a button to pause and resume the simulation, as in some of your earlier projects. Test it to make sure it works.

Exercise: Add a pair of buttons to add and remove energy to/from the system. A good way to do this is to multiply or divide all the velocities by 1.1. Again, test these buttons to make sure they work.

Question: Describe what happens when you continually remove energy from the system. How do the atoms arrange themselves? (Sketch a typical arrangement.)

Question: Describe what happens when you continually add energy to the system.

Exercise: Play with your program some more, perhaps trying different numbers of atoms, changing the width of the box, and using the buttons to add and remove energy. Describe at least one other interesting phenomenon that the simulated atoms exhibit.

Data output

What kind of data should you collect from this simulation? Energy is always a good place to start.

Exercise: Add `wtext` objects to your program to display the kinetic energy, potential energy, and total energy. Update these displayed values from your main simulation loop, once for each animation frame. To compute the kinetic energy, write a separate function that adds up the kinetic energies of all the atoms and returns the sum. To compute the potential energy, it's easiest to add a few lines of code to your `computeAccelerations` function, using a global variable to store the result so you can access it from your main loop. Be sure to include both the Lennard-Jones intermolecular potential energy and the "spring" potential energy associated with interactions with the walls ($\frac{1}{2}kx^2$, where k is the spring constant and x is the amount of compression). To be absolutely precise, you should add a small constant to the Lennard-Jones energy to compensate for the fact that you're setting the force to zero beyond a certain cutoff distance. After you insert all this code, check that the energy values are reasonable. About how much does the total energy fluctuate? (Because the energy can be positive or negative, please describe the fluctuation as an absolute amount, not as a percentage.) What about the kinetic and potential energies? What happens if you increase the value of `dt` to 0.025?

Another variable of interest is the temperature of the system. For a collection of N classical particles, the equipartition theorem tells us that each energy term that is quadratic in a coordinate or velocity (that is, each "degree of freedom") has an *average* energy of $\frac{1}{2}k_BT$, where k_B is Boltzmann's constant and T is the temperature. In two dimensions each molecule has only two translational degrees of freedom (v_x and v_y), so the average kinetic energy per molecule should be $2 \cdot \frac{1}{2}k_BT$. Thus, in natural units where $k_B = 1$, the temperature is precisely equal to the average kinetic energy per molecule.

In a macroscopic system with something like 10^{23} particles, the average kinetic energy per molecule wouldn't fluctuate measurably over time. In your much smaller system, the kinetic energy fluctuates quite a bit. To get a good value of the temperature, therefore, you need to average not only over all the molecules but also over a fairly long time period. This requires just a little more computation.

Exercise: Add code to your program to compute and display the average temperature of the system. You'll need a variable to accumulate the sum of the temperatures computed at many different times, and another variable to count how many values have been added into this sum so far. Increment these variables in your main loop (once per animation frame), then divide to get the average, and display this value using another `wtext` object. You'll need to reset both of these variables whenever energy is added to or removed from the system. Also add a button that manually resets the variables. Test your code and check that the results are reasonable.

In a similar way, you can also compute and display the average pressure of the system. In two dimensions, pressure is defined as force per unit *length*.

Exercise: Add code to your `computeAccelerations` function to compute the total outward force exerted on the walls of the box, and hence the instantaneous pressure of the system. Store the result in a global variable, and use that variable in your main simulation loop to compute and display the average pressure over time, just as you did for the temperature. To check that your results are reasonable, recall that in the limit of low density (where the distance between molecules is large compared to their sizes), the pressure is given by the ideal gas law: $P = Nk_BT/V$. (In two dimensions, V is actually an area.) Set `N` and `w` so the density of your system is reasonably low, add energy until the molecules are behaving like a gas, and compare the pressure of your system to the prediction of the ideal gas law. Show your data and calculations below.

Exercise: Add one more button to your program, to print the temperature, pressure, and total energy values (to the screen, separated by tabs) when it is pressed. This will allow you to record data for later analysis. Check that it works.

Congratulations! Your molecular dynamics simulation program is finished. Now would be a good time to clean up the code and add more comments if necessary. Then get ready to use your program for a systematic numerical "experiment." You'll be studying a system of a fixed number of atoms, in a fixed volume, over a wide range of temperatures. The number of atoms should be at least 200, but feel free to use more if your computer is fast enough. The volume should be large enough to give the atoms plenty of space: at least 10 units of volume per atom. Before you start to take data, fine-tune your program and check that everything is working correctly.

Exercise: Setting `N` to at least 200 and the volume to at least 10 units per atom, use your simulation to determine the energy and pressure as functions of temperature over a wide range of temperatures, from about 0.0001 to 1.0, with enough intermediate points to produce smooth graphs. Before recording each data point, be sure to let the system equilibrate long enough to give reasonably stable values of the temperature and pressure. Also please make some notes to describe the system's appearance at various temperatures. Copy your program's output into a spreadsheet and use the spreadsheet to plot graphs of E vs. T and P vs. T. Print the data and graphs and attach them to your lab report. Discuss these graphs in as much detail as you can, correlating them to your written notes. How does the pressure compare to the prediction of the ideal gas law? How does the system's heat capacity behave in the various temperature regions, and how does this behavior relate to what you learned about heat capacities in introductory physics?