Problem Set 2

Showing your work

Your homework assignments should show incremental development with at least two different commits over two different days of work. Assignments that do not show incremental development will incur a 5% penalty.

Reminder: No AI Tools

According to MSSE Department policy, use of AI tools is not permitted in Chem 274A. Do not use generative AI tools (e.g., ChatGPT, Claude, GitHub Copilot, or similar) for any part of this assignment, including planning, coding, or writing.

Contents

Section 1 - C++ Classes

We've seen some beginnings of a class representing a Molecule before. Now it is time to make that idea more complete, and add some useful features.

The file `molecule.cpp` contains the beginning of a molecule class. Your job is to add the following functionality:

Remember `const` correctness! Points will be deducted for functions that are not `const` correct, or if the code fails to compile.

- Default constructor (takes no arguments)
- A constructor that that takes in an existing `std::vector` of atoms
- Copy constructor
- A function to get an `Atom` from the class by index (and allows modification of the returned atom)
- A function to add an atom (appending to the end)
- A function that returns the number of atoms of the molecule (by convention called `size()`, returning a `size_t`)
- A function that removes all atoms (by convention called `clear()`)
- A function that computes & returns the molecular weight
- A function that computes & returns the center of mass (see below)
- A function that computes & returns the moment of inertia tensor (see below)

You are also required to write some tests demonstrating your code in the `main` function of that file. Some examples are shown there, but **you must add more** for the functions you have written.

Hint - storing the atoms internally using a `std::vector` will make this much easier

Center of mass

The center of mass is a 3d point the represents the mass-weighted center of the molecule.

$$C_x = \frac{\sum_i m_i x_i}{M} \; C_y \qquad = \frac{\sum_i m_i y_i}{M} \; C_z = \frac{\sum_i m_i z_i}{M}$$

where i represents the index of an atom in our molecule, and M is the total mass of the molecule. The function should return an `std::array` of three doubles.

Moment of inertia tensor

The moment of inertia tensor I is a common thing to compute for molecules. All you need is the coordinates and the atomic mass of each atom.

There is lots of info online about these tensors - see here for an example.

The moment of inertia tensor is represented as a 3×3 symmetric matrix

$$\mathbf{I} = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

where the diagonal elements are given by

$$I_{xx} = \sum_i m_i (y_i^2 + z_i^2) \; I_{yy} \qquad = \sum_i m_i (x_i^2 + z_i^2) \; I_{zz} = \sum_i m_i (x_i^2 + y_i^2)$$

and off diagonal elements

$$I_{xy} = -\sum_i m_i x_i y_i \; I_{xz} \qquad = -\sum_i m_i x_i z_i \; I_{yz} = -\sum_i m_i y_i z_i$$

where i represents the index of an atom in our molecule. Since this is a symmetric matrix, $I_{xy} = I_{yx}$ and so on.

Since we are using C++, which has 0-based indexing, then `x=0`, `y=1`, and `z=2`. Therefore, I_{xx} would be stored in a variable such as `I[0][0]`, and I_{yz} would be stored in `I[1][2]`.

Write a function (part of the Molecule class) that computes and returns the moment of inertia tensor. Since we know the size of this matrix and it is always 3×3, you can return a nested `std::array` object.

Consider whether you should store this in your class or always compute this on the fly. Think about the other functions you wrote, and what happens to a stored tensor if atoms are added/removed.

Section 2 - Python Inheritance and Composition.

The background for this assignment is long, but you should only have to write a few lines of code to complete these tasks (probably less than 20).

Molecular Dynamics Simulations - Background

In our last homework, we wrote a class to describe the movement of a harmonic oscillator through time. The equations for position and velocity were obtained by solving the differential equation for the harmonic oscillator constructed using Newton's second law (F=ma) and the force on the harmonic oscillator (F=kx).

The harmonic oscillator is a rare example of a system where such an exact solution exists. However, in molecular dynamics simulations, we are often interested in more complicated systems for which there is not an analytical solution to describe the movement.

In molecular dynamics simulations, the movement of molecules is simulated by calculating the forces on atoms and updating their positions based on those forces. This is repeated many times (millions!) to obtain the trajectory through time of a molecular system. The output of a molecular dynamics simulation consists of a record of coordinates (a trajectory) for different timepoints, and information about the energy of the system at those timepoints. The trajectory can be analyzed to predict molecular properties or behavior, and visualized to show a model of the movement of a system through time.

Force is equal to the negative gradient of potential energy. MD simulations use a potential energy function and Newton's second law (the force on an object is equal to the object's mass times its acceleration) to calculate positions of atoms.

$$\vec{F} = -\nabla U$$

$$\vec{F} = m \cdot \vec{a} = m \cdot \frac{d\vec{v}}{dt} = m \cdot \frac{d^2\vec{r}}{dt^2}$$

By setting these two equations equal to one another, we can obtain an expression for the acceleration, which will allow us to calculate updated positions based on initial coordinates and velocities.

$$\overrightarrow{a_i}(t_0) = \frac{1}{m}\overrightarrow{F_i}(t_0) = -\frac{1}{m_i}(\nabla U)_i$$

Given initial conditions (positions and velocities), we can calculate positions ($\vec{x}$) based on previous positions and acceleration and an amount of time (Δt).

There are many algorithms for computing trajectories from a potential energy and molecular system. In this homework, you will implement at least two molecular dynamics integration algorithms (the algorithms which are used to propagate the movement of system in time forward.)

You will use the diatomic object written in your last homework assignment to test your integrators. Usually in a molecular system, there will be many more atoms and the potential energy function will be more complicated and writing an analytical equation for the positions of the atoms is not possible. The diatomic class is described by a potential energy equation which is commonly used in molecular dynamics simulations to describe bonds (the harmonic oscillator) and has the benefit of having an analytical solution, so you can directly compare the behavior predicted by the analytical solution and the molecular dynamics integrator.

Starting Code

For this homework, you implement the Velocity Verlet algorithm and at least one other algorithm of your choice using inheritance. The algorithms you can choose from are at the end of this homework. You have been provided with starter code for this task:

1. `diatomic.py` - Contains diatomic class (`Diatomic`) which has methods implemented in the Specifications of Problem Set 1. You will add two methods to this class (as outlined in Specifications, which will make MD simulation possible)
2. `integrator.py` - Contains `IntegratorBaseClass` class. This class contains a constructor, run simulation, and update trackers method that will be common to all integrators.
3. `run.py` - contains a script which will run a Velocity Verlet simulation and create a plot comparing positions predicted by Velocity Verlet to analytical values.

Specifications

1. Starting with the provided `Diatomic` class from Problem Set 1, add a method called `force` to compute the force based on position. Remember that for a harmonic oscillator, $F(x) = -kx$. This method should **return** the force.

2. Add another method to your `Diatomic` class called `acceleration`. This method should calculate the acceleration based on the force (using your `force` method from the first step). This should use the equation `F=ma` (ie, $a = \frac{F}{m}$). This method should also **return** the calculated acceleration. **This acceleration will be used in many of the integration algorithms.**

3. Now you have all the pieces for your integrators. For each integration method (you should do Velocity Verlet and one other method of your choice), write a class which inherits from `IntegratorBaseClass`. Write the `step` method. After writing the `step` method for the Velocity Verlet integration method, you should be able to run `run.py` and obtain graphs comparing the integration method and the analytical solution. **You do not need to change any code in IntegratorBaseClass or run.py to implement your integration methods, and your VelocityVerlet method should only need a `step` method.**

   Notice that the `IntegratorBaseClass` takes an instance of the diatomic class in the constructor. This is an example of an object oriented programming concept called *composition*. The integrator has another object (the diatomic) as a component. The `run` methods in your integrator classes should update the state (position and velocity) of the diatomic for each time step (ie, `self.diatomic.position = SOMETHING`)

4. Compare the trajectory predicted by your integration methods to the analytical position values. **Hint** - If your integration is implemented correctly, you will see *very good* agreement between Velocity Verlet and the analytical solution for the parameters provided in the run script.

Molecular Dynamics Integrators

This section shows several molecular dynamics algorithms. Each of these algorithms has advantages / disadvantages concerning accuracy and stability which we will not have time to cover. Use this section as a reference as you are completing the tasks in the `Specifications` section.

Velocity Verlet Integrator

Perhaps the most commonly used algorithm is the **Velocity Verlet** (pronounced ver-lay) algorithm. Everyone will implement a Velocity Verlet algorithm.

$$\overrightarrow{x_i}(t_0 + \Delta t) = \overrightarrow{x_i}(t_0) + \overrightarrow{v_i}(t_0)\Delta t + \frac{1}{2}\overrightarrow{a_i}(t_0)\Delta t^2$$

$$\overrightarrow{v_i}(t_0 + \Delta t) = \overrightarrow{v_i}(t_0) + \frac{\overrightarrow{a_i}(t_0) - \overrightarrow{a_i}(t_0 + \Delta t)}{2}\Delta t$$

You can either use the formulas above, or in half steps (as described below). A nice explanation of the steps to implement the half step algorithm can be found on Wikipedia. It is a bit nicer than using the equations above beause it doesn't rely on the past state of the sytem.

1. Calculate the velocity at a half step:

$$\vec{v}(t + \frac{1}{2}\Delta t) = \vec{v}(t) + \frac{1}{2}\vec{a}(t)\Delta t$$

2. Calculate the updated position:

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t + \frac{1}{2}\Delta t)\Delta t$$

3. Calculate the updated velocity using half step velocity from (1) and updated acceleration based on position in step 2.

$$\vec{v}(t + \Delta t) = \vec{v}(t + \frac{1}{2}\Delta t) + \frac{1}{2}\vec{a}(t + \Delta t)\Delta t$$

Euler Integrator

The Euler integrator is the simplest integrator. However, it is not time reversible and generally does a poor job of maintaining constant energy, so it is not usually used.

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t)\Delta t$$

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t + \Delta t)\Delta t$$

Verlet Integrator

The Velocity Verlet algorithm discussed previously is an improvement on the Verlet algorithm. The Verlet algorithm is one of the earliest molecular dynamics integrators used. It is more difficult to implement than the Euler or Velocity Verlet integrator because it is not "self-starting". Calculation of the first step is different than every other step and the next position relies on the past state of the simulation.

First Step

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

$$\vec{v}(t+\Delta t) = \frac{\vec{x}(t+\Delta t) - \vec{x}(t)}{\Delta t}$$

All other steps

$$\vec{x}(t+\Delta t) = 2\vec{x}(t) - \vec{x}(t-\Delta t) + \vec{a}(t)\Delta t^2$$

In the Verlet method, the velocity is not actually used to calculate the position. However, it would be necessary to calculate observables like the kinetic energy. It is approximated at a particular time using the previous position and the next position.

$$\vec{v}(t+\Delta t) = \frac{\vec{x}(t+\Delta t) - \vec{x}(t-\Delta t)}{2\Delta t}$$

A question to ponder - Why does the velocity calculation for all steps besides step 1 have a 2 in the denominator?

Beeman Integrator

The Beeman algorithm is very similar to the Verlet algorithm. Like the Verlet algorithm, the Beeman algorithm is not self-starting. It is also harder to implement than either the Velocity Verlet or Euler methods.

First Step

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

$$\vec{v}(t+\Delta t) = \frac{\vec{x}(t+\Delta t) - \vec{x}(t)}{\Delta t}$$

All other steps

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \vec{v}(t)\Delta t + \frac{1}{6}(4\vec{a}(t) - \vec{a}(t-\Delta t))\,\Delta t^2$$

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{1}{6}(2\vec{a}(t+\Delta t) + 5\vec{a}(t) - \vec{a}(t-\Delta t))\,\Delta t$$

Reflection and Documentation

Create a `Makefile` which runs each integrator and saves a plot of the integrator results with the analytical results. Your make file has two required targets: `VelocityVerlet` (runs Velocity Verlet simulation) and another target for your integrator of choice. Make the name of this target `Integrator2`.

Like always, you should include a `README.md` that includes information about the homework and how to use your mMkefile.

Write the answer to these questions in your README for this assignment.

Answering Reflection Questions

Your written answers should be *complete explanations*, not short phrases. A complete answer:

- References the code or class design from the assignment (e.g., mention specific classes, methods, or objects).
- Explains the reasoning, not just the fact (say *why* inheritance or composition is used, not only that it is).
- Uses correct terminology from object-oriented programming.
- Is written in full sentences and paragraphs.

Grading: Full credit requires answers that connect to code and explain *why*. Answers that only identify a concept without explanation will receive partial credit.

1. This homework uses both **inheritance** and **composition**. Where are each used and how? Why?
2. Discuss the advantages and disadvantages of using inheritance for this task. Then, do the same for composition. How do these two concepts complement each other in this problem?
3. Suppose you wanted to add a method to all integrators that prints the system state (current position, velocity, kinetic energy, potential energy). Where would you put this method, and what object-oriented programming concept would you be using to make it available to every integrator?
4. Now suppose you wanted to simulate a molecular system other than a diatomic (for example, a triatomic or more general molecule). Which OOP concept makes this extension straightforward, and why?
5. Conceptually, what is the difference between the analytical prediction of the movement of our harmonic oscillator and the one predicted using the molecular dynamics simulation? **Hint** - The answer is *not* that an MD simulation is "more realistic". Both are applications of the harmonic oscillator model.