# Problem Set 2

## Contents



## Section 1 - C++ Classes

The file `molecule.cpp` contains the beginning of a molecule class. Add the following member functions to the class.

Remember const correctness! Some functions should be const.

- Default constructor (takes no arguments)
- Constructor that takes in an existing std::vector of atoms
- A function to get an `Atom` from the class by index
- A function to add/append an `Atom` instance.
- A function that prints out the size (number of atoms) of the molecule (called `size()`, returning a `size_t`)
- A function that prints out the contents of the molecule (each atom being on its own line)
- A destructor that simply prints that it is destructing and how many atoms it is destructing.

## Section 2 - Python Inheritance

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 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 the movement of a system through time. It is important to emphasize that the trajectories generated do not necessarily represent the "true movement" of a molecular system. Molecular dynamics is a statistical prediction method.

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.

$$ \vec{a}_{i}(t_0) = \frac{1}{m}\vec{F}_{i}(t_0) = -\frac{1}{m}_{i}(\nabla U)_{i} $$

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

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 propogate the movement of system in time foward.)

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.

$$ \vec{x}*i* (*t*{0} + \Delta t) = \vec{x}*i* (*t*{0}) + \vec{v}*{i}* (*t*{0}) \Delta t + \frac{1}{2} \vec{a}*{i}*(*t*{0}) \Delta t^{2} $$

$$ \vec{v}*i* (*t*{0} + \Delta t) = \vec{v}*i* (*t*{0}) + \frac{\vec{a}*{i}*(*t*{0}) - \vec{a}*{i}*(*t*{0}+\Delta t)}{2} \Delta t $$

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:

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

2. Calculate the updated position:

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

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.

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

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

**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 difficulat to implement than the Euler or 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**

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

**All other steps**

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

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.

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

**Beeman Integrator**

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

**First Step**

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

**All other steps**

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

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

## 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 three required targets: `lint` (runs black and flake8), `VelocityVerlet` (runs Velocity Verlet simulation) and another target for your integrator of choice.

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

1. Which integrator did you choose to implement and why?
2. Imagine that you wanted to add a method to all of the integrators that allowed you to print the system state (current position, velocity, kinetic energy, potential energy). How would you do that? Where would you add your method, and what would you call the method?
3. Do you see the advantage of using inheritance and object oriented programming for this task? If so, what do you perceive the advantage to be? If not, how do you think the structure of the code could be improved?