Software Engineering in Molecular Science - Taking Monte Carlo Simulation of Lennard-Jones Fluids as an Example

Contents

Chapter 1. Introduction: MC Simulation on Lennard-Jones Fluids

1-1. Scope of the documentation

This documentation aims to be a useful hands-on tutorial on building a software package in Python based on the molecular mechanics (MM) project introduced in 2019 software engineering summer school held by Molecular Science Software Institute (MolSSI). To help readers build a solid foundation required to complete the project, we also integrate parts of the materials covered in several Udemy online courses, including

(1) *How to Create, Publish, Maintain andContribute to Opensource Software:*
https://www.udemy.com/python-awesome-tools/

(2) *Python beyond Basics - Object-Oriented Programming (OOP):*
https://www.udemy.com/python-beyond-the-basics-object-oriented-programming/

(3) *Parallel Computing with HPC Systems:*
https://www.udemy.com/learn-to-use-hpc-systems-and-supercomputers/

Ideally, this documentation will guide the reader to finish the molecular mechanics (MM) project at MolSSI summer school, from setting up the system, creating a GitHub repository, refactoring the code, accelerating the computation through C++ binding and parallel computing, writing testing codes, performing unit testing and continuous integration and organizing a complete Python documentation. By following the instructions about building a Python package for Monte Carlo simulation on Lennard-Jones fluids in this documentation, the reader should be able to realize the fundamentals of software engineering and cultivate the ability to develop a Python package. In spite of the minimum prerequisite of the project include basic understanding about Python coding and Metropolis-Hasting algorithm which will be used in Monte-Carlo simulation, familiarity with fundamentals of statistical thermodynamics and molecular science might facilitate a deeper understanding about the project. (Note: The remaining of this chapter is basically the same as the introductory material at MolSSI summer school written by Dr. Eliseo Marin-Rimoldi and Dr. John D. Chodera, which can also be obtained from the GitHub repository of this documentation: https://github.com/wehs7661/MCLJ_software)

1-2. A quick review of the required fundamentals of the project

(1) Monte Carlo integration
In statistical mechanics, we are interested in computing averages of thermodynamic properties as a function of atom positions and momenta. A thermodynamic average depending only on configurational properties can be computed using the following expectation value integral

$$\langle Q \rangle = \int_V Q(\mathbf{r}^N)\rho(\mathbf{r}^N)d\mathbf{r}^N \tag{1}$$

$\mathbf{r}^N$ is a $3N$ dimensional vector containing the positions of the N atoms, where $Q(\mathbf{r}^N)$ is thermodynamic quantity (partition function) of interest that depends only on the configuration $\mathbf{r}^N$, $\rho(\mathbf{r}^N)$ is the probability density whose functional form depends on the statistical mechanical ensemble of interest, and V defines the volume of configuration space over which ρ has support. Note that the integrals over momenta have been factored out, as they can be evaluated analytically. The integral (Eqn. 1) is very hard to compute even for small atomic systems. For instance, a monoatomic system of 10 atoms leads to a 30-dimensional integral. Consequently, we need to resort to a numerical integration scheme if we want to study atomic systems.

Monte Carlo methods are numerical techniques frequently used to estimate complex multidimensional integrals which otherwise could not be performed. For instance, the integral of the function $f(x)$, where $x \in R^M$, is approximated as

$$I = \int_V f(\mathbf{x})d\mathbf{x} = \int_V \frac{f(\mathbf{x})}{h(\mathbf{x}}) h(\mathbf{x})d\mathbf{x} = \left\langle \frac{f(\mathbf{x})}{h(\mathbf{x})} \right\rangle_{h(\mathbf{x})} \tag{2}$$

The idea of Monte Carlo integration is to estimate the expectation value $\left\langle \frac{f(\mathbf{x})}{h(\mathbf{x})} \right\rangle_{h(\mathbf{x})}$ by generating random samples of $\mathbf{x}$ from the probability density $h(\mathbf{x})$

(2) Importance sampling
In Equation 2, we are free to chose the probability distribution $h(\mathbf{x})$. The simplest case is to uniformly generate $\mathbf{x}$ in the volume V. In this way, $h(\mathbf{x})$ becomes constant as

$$h(\mathbf{x}) = \frac{1}{V} \tag{3}$$

Using this sampling density $h(\mathbf{x})$, the integral (Eqn. 2) becomes

$$I = \int_V f(\mathbf{x})\, d\mathbf{x} \approx \frac{V}{N} \sum_{i=1}^{N} f(\mathbf{x}_i) \tag{4}$$

where N is the total number of random samples and $f(\mathbf{x}_i)$ is the integrand evaluated using the i^{th} sample. While using a uniform sampling density often works sufficiently well for simple unidimensional cases, it generally fails to produce useful estimates for complex problems.

The problem at hand involves the evaluation of $3N$-dimensional integral Eqn. 1, which is dominated by a small region of configuration space. Using a uniform probability distribution $h(\mathbf{r}^N)$ over the configuration space hypervolume V^{3N} to generate representative samples of this subset is not efficient, as most states generated this way would have a low weight.

A solution to this problem is to sample positions $\mathbf{r}^N$ from the desired equilibrium probability density $\rho\left(\mathbf{r}^N\right)$:

$$\mathbf{r}^N \quad \sim \quad \rho(\mathbf{r}^N) \tag{5}$$

This is a way to generate relevant configurations more frequently than configurations that have low probability. Mathematically, we set $h(\mathbf{r}^N) = \rho\left(\mathbf{r}^N\right)$. This idea is known as *importance sampling*. Combining Eqn. 1 and Eqn. 2 and the condition $h(\mathbf{r}^N) = \rho\left(\mathbf{r}^N\right)$, we find that

$$\langle Q \rangle \approx \frac{1}{N} \sum_{i=1}^{N} Q\left(\mathbf{r}_i^N\right). \tag{6}$$

Thus, we can get thermodynamic properties by simply computing an unweighted sample average, given that we perform importance sampling from $h(\mathbf{r}^N) = \rho\left(\mathbf{r}^N\right)$.

(3) Detailed balance
The question now becomes how to generate such atomic positions $\mathbf{r}^N$ (or *states*) distributed according to $\rho\left(\mathbf{r}^N\right)$. In 1953, Metropolis, Rosenbluth, Rosenbluth, and Teller introduced a solution based on Markov chains. They proposed to use the detailed balance condition in order to ensure proper configurational sampling from the statistical mechanical distribution of interest. In order to generate a new configuration n from an old configuration m, the detailed balance condition is

$$\rho_m\left(\mathbf{r}^N\right) \alpha\left(m \to n\right) P_{acc}\left(m \to n\right) = \rho_n\left(\mathbf{r}^N\right) \alpha\left(n \to m\right) P_{acc}\left(n \to m\right) \tag{7}$$

Where $\rho_m\left(\mathbf{r}^N\right)$ is the probability of observing state m, $\alpha\left(m \to n\right)$ is the probability of attempting to generate a new state n starting from a state m and $P_{acc}\left(n \to m\right)$ is the probability of accepting such transition. Basically, the condition of detailed balance tells us that the "flux" of transitions from state m to state n equals the flux from state n to state m at equilibrium.

There are many ways to satisfy Eqn. 7 by construction of different acceptance probabilities. While Metropolis et. al proposed a choice that both satisfies Eqn. 7 and maximizes the average acceptance

probability $\langle P_{acc} \rangle$, Hastings generalized this to the case where proposal probabilities are not symmetric, such that $\alpha(m \to n) \neq \alpha(n \to m)$, producing the acceptance criteria:

$$P_{acc}(m \to n) = \min \left[1, \frac{\alpha(n \to m)}{\alpha(m \to n)} \frac{\rho_n\left(\mathbf{r}^N\right)}{\rho_m\left(\mathbf{r}^N\right)} \right] \tag{8}$$

This algorithm is one of a general class of *Markov chain Monte Carlo (MCMC)* algorithms that generate Markov chains to sample a desired target density, and a great deal of the MCMC literature is valuable for molecular simulations.

1-3. Monte-Carlo simulation of the Lennard-Jones fluid in the canonical ensemble

Assume we have N monoatomic particles that interact using the Lennard-Jones (LJ) pairwise potential:

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{9}$$

where r is the interparticle distance, σ is the distance where the interaction energy is zero, and ϵ is the well depth. For simulating argon, for example, a common choice is $\sigma = 3.4$ Åand $\epsilon/k_B = 120$ K.

Our goal is to generate a set of states of N LJ particles distributed according to the canonical (NVT) ensemble

$$\rho_n\left(\mathbf{r}^N; \beta\right) = Z(\beta)^{-1} e^{-\beta U\left(\mathbf{r}^N\right)} \tag{10}$$

$$Z(\beta) \equiv \int_V e^{-\beta U\left(\mathbf{r}^N\right)} d\mathbf{r}^N \tag{11}$$

where $U\left(\mathbf{r}^N\right)$ is the potential energy of the system, $\beta = (k_B T)^{-1}$ is the inverse temperature, k_B is the Boltzmann constant, and T is the absolute temperature.

Note that $U\left(\mathbf{r}^N\right)$ is given by

$$U\left(\mathbf{r}^N\right) = \sum_{i<j} U\left(r_{ij}\right) \tag{12}$$

where $r_{ij} \equiv ||\mathbf{r}_j^N - \mathbf{r}_i^N||_2$ is the interparticle separation distance.

Substituting Eqn. 10 into Eqn. 8 and assuming $\alpha(n \to m) = \alpha(m \to n)$, we obtain

$$P_{acc}(m \to n) = \min\left[1, e^{-\beta \Delta U}\right] \tag{13}$$

where $\Delta U \equiv U(\mathbf{r}_m^N) - U(\mathbf{r}_n^N)$ is the difference in potential energy of the system between the new state n and the old state m. Note that the argument of the energy $\mathbf{r}^N$ has been dropped for clarity.

(1) Flow of Calculations in a Metropolis Monte Carlo simulation
The following workflow can be used to implement the Metropolis algorithm to sample the canonical ensemble of configurations of LJ particles:

- Generate an inital system state m.
- Choose an atom with uniform probability from $\{1, \ldots, N\}$ from old state m.
- Propose a new state n by translating a LJ particle by a uniform random displacement $\Delta r \sim U(-\Delta x, +\Delta x)$ in each dimension. The displacement scale Δx should not be too large as this would likely result in particle overlaps, but should not be too small as this would result in a slow sampling of configurational space. More on this below.
- The difference in energy between the new and old states is computed. Note that you do not need to compute the *total* system energy difference, as all particles but one remain at the same position. It is enough to get difference in energy of the selected molecule in the new and old states.

- The new state is accepted or rejected using the Metropolis criterion. Practically, this can be implemented as follows. If a move from m to n is "downhill", $\beta\Delta U \leq 0$, the move is always accepted. For "uphill" moves, a random number ζ is generated uniformly on (0,1). If $\zeta < \exp[-\beta\Delta U]$, the move is accepted. Otherwise, the move is rejected. If a non-symmetric proposal is used, this acceptance scheme will have to be modified to implement Eqn. 8.

(2) Technical considerations

- Initial configuration
  To start the simulation, we have to generate an initial configuration. We will provide two options: start using a random configuration or start using a previously equilibrated state. During the Metropolis Monte Carlo simulation, particle translations will help to create configurations consistent with the temperatures and box volumes that we set and will remove any overlap that might exist in the initial random configuration. In the second option, we will use a pre-equilibrated NIST configuration that will help us benchmark the energy calculations of our code.
- Random number generation
  Computers cannot generate truly random numbers. Instead, they rely on *pseudorandom number generators* (PRNGs) that aim to produce random numbers with the desired statistical properties and long recurrence times between repeats of the same random number sequence. Using a low-quality random number generator can lead to simulation artifacts that can lead to incorrect physical behavior. We recommend you avoid writing your own PRNGs, as this can lead to inadvertent implementation of an ill-conceived algorithm. Instead, rely on high-quality, well-understood PRNGs and implementations that have are well-supported, such as the Mersenne Twister implementation provided by `numpy.random`.
- Equilibration
  When the initial configuration is highly atypical compared to true samples from the equilibrium density—such as the initial 3D lattice conditions compared to a true disordered liquid state—it may require very long simulation times for the bias in equilibrium averages computed over the entire trajectory to become small compared to the statistical error. It is therefore common practice to discard some initial part of the simulation to *equilibration* and to average over the subsequent *production* region to minimize this bias at the cost of potentially increasing statistical error by including less data in the average. While common practice traditionally had selected an arbitrary initial portion of the simulation to equilibrium, modern best practice recommends the use of an automated approach for selecting the optimal equilibration/production split point in a manner that maximizes the number of statistically uncorrelated samples in the production part of the trajectory. To do this, you can use the `pymbar.timeseries.detectEquilibration` function from the `pymbar` module to analyze an array containing timeseries data for your observable (such as energies, box volumes, or densities).
- System size and periodic boundary conditions
  A typical simulation of a Lennard-Jones fluid is carried out anywhere from 216 to 10,000 particles. This amount of particles is far away of being representative of a bulk liquid. To get around this problem, we employ a trick called periodic boundary conditions. The primary simulation box is surrounded by copy images of itself. For a cubic simulation, there would be 26 images around the central box. This trick has two implications
  - If the position of a particle (i.e. Cartesian coordinates) is outside the simulation box after a particle translation, an identical particle should enter the box through the opposite face of the box. As shown in Figure 1., molecule one is displaced outside the bounds of the central box and placed back in through the oposite side of the box.
  - When computing distances r_{ij} used in the evaluation of the LJ potential (Eqn. 12), we use the *minimum image distance*. As shown in Figure 2., Molecule 1 does not interact with molecule 4 located in the simulation box because the distance between particles is greater than the cutoff. However, molecule 1 does interact with the image of molecule 4 located in box E.

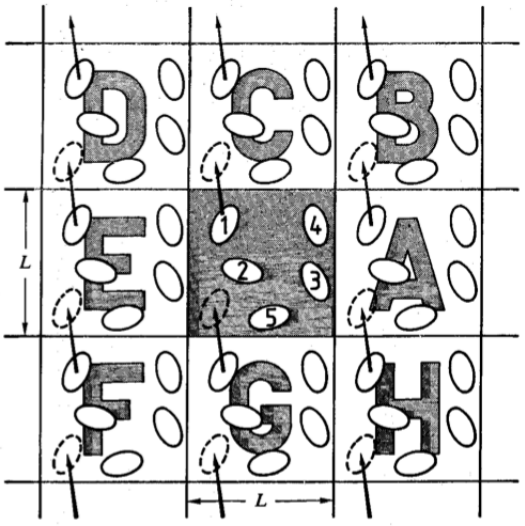

Figure 1: Periodic boundary condition

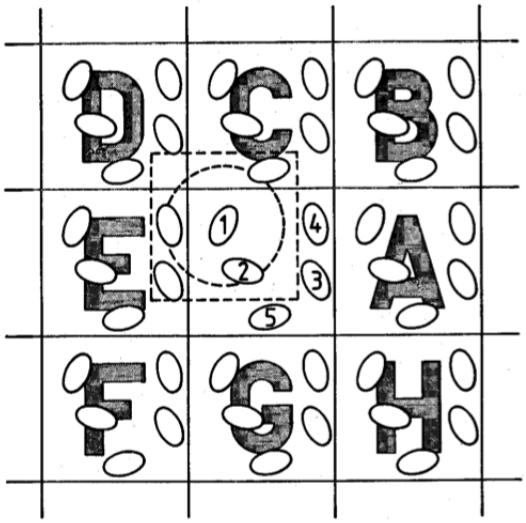

Figure 2: Minimum image distance.

- Maximum displacement Δx
  As noted above, the Metropolis Monte Carlo algorithm requires translating a selected LJ particle by a random perturbation. This displacement should not be too large as this would result in particle overlaps and low acceptance rates; on the other hand, it should not be so small as to result in inefficient sampling of configuration space. A common practice is to adjust the maximum particle displacement Δx during an explicit equilibration phase in order to achieve $\sim$50% acceptance translation rates over a recent window of $\sim N$ Monte Carlo trial moves.

- Energy truncation and tail corrections
  If two particles are separatedby more than a certain distance, we typically truncate their interaction energy if $r > r_c$, where r_c denotes the *cutoff distance*. Truncating interactions removes contribution to the potential energy that might be non negligible and can lead to significant artifacts, such as significantly perturbed densities when a barostat is used to sample the NPT ensemble due to neglected long-range dispersion interactions. We can estimate the truncated interactions by incorporating an energy correction, known as the tail or long range correction. For the Lennard-Jones fluid, we assume that we have an homogeneous liquid at $r > r_c$ to obtain the correction for neglecting this contribution for all interacting pairs of particles:

$$U_{\text{correction}} = \frac{8\pi N^2}{3V}\epsilon\sigma^3\left[\frac{1}{3}\left(\frac{\sigma}{r_c}\right)^9 - \left(\frac{\sigma}{r_c}\right)^3\right] \tag{14}$$

  For a Lennard-Jones fluid, it is common to set $r_c \sim 3\sigma$, since the pair interaction at this separation is small, $U(3\sigma) \approx 4[(3)^{-12} - (3)^{-6} \approx -0.0055\epsilon$, or about 0.5% of the well depth ϵ. You will want to verify that your computed properties are relatively insensitive to the choice of cutoff r_c so that a too-short cutoff does not induce artifacts in computed physical properties.

- Reduced units
  Lennard-Jones fluids have the surprisingly pleasant behavior of possessing universal behavior when expressed in terms of *reduced units* as

$$U^*\left(r_{ij}\right) = 4\left[\left(\frac{1}{r^*_{ij}}\right)^{12} - \left(\frac{1}{r^*_{ij}}\right)^{6}\right] \tag{15}$$

  where

$$U^* = \frac{U}{\epsilon} \tag{16}$$

and

$$r^* = \frac{r}{\sigma} \tag{17}$$

That is, when plotted in reduced units, all Lennard-Jones fluids exhibit the same universal behavior despite the exact choices of ϵ and σ used in the simulation. See the following table for a list of variables in reduced units. Using reduced units for input and output will allow you to compare your results with others.

Quantity	Expression	Quantity	Expression
Length	$L^* = L/\sigma$	Temperature	$T^* = k_B T/\epsilon$
Density	$\rho^* = N\sigma^3/V$	Volume	$V^* = V/\sigma^3$
Energy	$U^* = U/\epsilon$	Time	$t^* = t\sqrt{\frac{\epsilon}{m\sigma^2}}$
Pressure	$P^* = P\sigma^3/\epsilon$	-	-

1-4. The goal of the molecular mechanics (MM) project at 2019 MolSSI summer school

There are several tasks to be completed in the MM project at 2019 MolSSI summer school, including implementing one or more methods to generate initial configurations, the pairwise and long tail correction equations, a function that computes the total energy of the system and the Metropolis algorithm and comparing to the NIST benchmark (https://mmlapps.nist.gov/srs/LJ_PURE/mc.html). As mentioned, the initial configuration for Monte-Carlo simulation could be either randomly generated or the pre-equilibrated configuration provided by NIST (see https://bit.ly/31iESca). To compare to the NIST benchmark for thermodynamic data, in this documentation, we will start the simulation with the state of $T* = 0.9$, $\rho* = 0.9$, and $r_c = 3\sigma$.

1-5. Project-based section: MC simulation code to start with

A Python code named `MCLJ_original.py` for the Monte-Carlo simulation of Lennard-Jones fluid to start with is shown as follows. In the following chapters, in addition to refining the code by refactoring with different design patterns, C++ binding and parallelization, we will guide the reader develop a software package based on this code and make it as an open source and all the relevant files can be obtained from the GitHub repository of this documentation: https://github.com/wehs7661/MCLJ_software.

```
import os
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from mpl_toolkits.mplot3d import Axes3D


# Generate initial state
def generate_initial_coordinates(method='random', file_name=None,
    num_particles=None, box_length=None):

    if method is 'random':
```

```
        coordinates = (0.5 - np.random.rand(num_particles, 3))   box_length

    elif method is 'file':
        coordinates = np.loadtxt(file_name, skiprows=2, usecols=(1, 2, 3))

    return coordinates


# Lennard Jones potential implementation
def lennard_jones_potential(rij2):

    sig_by_r6 = np.power(1 / rij2, 3)
    sig_by_r12 = np.power(sig_by_r6, 2)
    return 4.0   (sig_by_r12 - sig_by_r6)


# Minimum image distance implementation
def minimum_image_distance(r_i, r_j, box_length):

    rij = r_i - r_j
    rij = rij - box_length   np.round(rij / box_length)
    rij2 = np.dot(rij, rij)
    return rij2


# Computation of the total system energy
def calculate_total_pair_energy(coordinates, box_length, cutoff2):

    e_total = 0.0
    particle_count = len(coordinates)

    for i_particle in range(particle_count):
        for j_particle in range(i_particle):
            r_i = coordinates[i_particle]
            r_j = coordinates[j_particle]
            rij2 = minimum_image_distance(r_i, r_j, box_length)
            if rij2 < cutoff2:
                e_pair = lennard_jones_potential(rij2)
                e_total += e_pair

    return e_total


def calculate_tail_correction(box_length, cutoff, number_particles):

    volume = np.power(box_length, 3)
    sig_by_cutoff3 = np.power(1.0 / cutoff, 3)
    sig_by_cutoff9 = np.power(sig_by_cutoff3, 3)
    e_correction = sig_by_cutoff9 - 3.0   sig_by_cutoff3
    e_correction  = 8.0 / 9.0   np.pi   number_particles / volume
   number_particles

    return e_correction

```

```
66
67 def get_particle_energy(coordinates, i_particle, cutoff2):
68
69     e_total = 0.0
70     i_position = coordinates[i_particle]
71
72     particle_count = len(coordinates)
73
74     for j_particle in range(particle_count):
75         if i_particle != j_particle:
76
77             j_position = coordinates[j_particle]
78             rij2 = minimum_image_distance(i_position, j_position, box_length)
79
80             if rij2 < cutoff2:
81                 e_pair = lennard_jones_potential(rij2)
82                 e_total += e_pair
83
84     return e_total
85
86
87 def accept_or_reject(delta_e, beta):
88
89     if delta_e <= 0.0:
90         accept = True
91     else:
92         random_number = np.random.rand(1)
93         p_acc = np.exp(-beta delta_e)
94         if random_number < p_acc:
95             accept = True
96         else:
97             accept = False
98
99     return accept
100
101
102 def adjust_displacement(n_trials, n_accept, max_displacement):
103     acc_rate = float(n_accept) / float(n_trials)
104     if acc_rate < 0.380:
105         max_displacement  = 0.8
106     elif acc_rate > 0.42:
107         max_displacement  = 1.2
108     n_trials = 0
109     n_accept = 0
110     return n_trials, n_accept, max_displacement
111
112
113 if __name__ == "__main__":
114
115     start = time.time()
116
117     # ------------------
118     # Parameter setup
119     # ------------------
```

```
120
121     reduced_temperature = 0.9
122     max_displacement = 0.1
123     n_steps = 1000000
124     freq = 1000
125     tune_displacement = True
126     simulation_cutoff = 3.0
127     element = 'C'
128     beta = 1 / reduced_temperature
129
130     # ------------------------------
131     # Coordinate initialization
132     # ------------------------------
133
134     # Method = random
135     build_method = 'random'
136     reduced_density = 0.9
137     num_particles = 500
138     box_length = np.cbrt(num_particles / reduced_density)
139     coordinates = generate_initial_coordinates(
140         method=build_method, num_particles=num_particles, box_length=
    box_length)
141
142     # ------------------------------
143     # Simulation initialization
144     # ------------------------------
145
146     # Method = file
147     # build_method = 'file'
148     # file_name = os.path.join('..', 'nist_sample_config1.txt')
149     # coordinates = generate_initial_coordinates(method=build_method,
    file_name=file_name)
150     # num_particles = len(coordinates)
151     # with open(file_name) as f:
152     #   f.readline()
153     #   box_length = float(f.readline().split()[0])
154
155     simulation_cutoff2 = np.power(simulation_cutoff, 2)
156     n_trials = 0
157     n_accept = 0
158     energy_array = np.zeros(n_steps)
159
160     total_pair_energy = calculate_total_pair_energy(
161         coordinates, box_length, simulation_cutoff2)
162     tail_correction = calculate_tail_correction(
163         box_length, simulation_cutoff, num_particles)
164
165     traj = open('traj.xyz', 'w')
166
167     # ------------------------------------
168     # Metropolis Monte Carlo algorithm
169     # ------------------------------------
170
171     for i_step in range(n_steps):
```

```
172
173         n_trials += 1
174
175         # ------------------------------
176         #  Propose a Monte Carlo Move
177         # ------------------------------
178         i_particle = np.random.randint(num_particles)
179         random_displacement = (
180             2.0   np.random.rand(3) - 1.0)   max_displacement
181
182         current_energy = get_particle_energy(
183             coordinates, i_particle, simulation_cutoff2)
184
185         # Make a copy before adding random displacement
186         proposed_coordinates = coordinates.copy()
187         proposed_coordinates[i_particle] += random_displacement
188         proposed_energy = get_particle_energy(
189             proposed_coordinates, i_particle, simulation_cutoff2)
190
191         delta_e = proposed_energy - current_energy
192
193         accept = accept_or_reject(delta_e, beta)
194
195         if accept:
196             total_pair_energy += delta_e
197             n_accept += 1
198             coordinates[i_particle] += random_displacement
199
200         total_energy = (total_pair_energy + tail_correction) / num_particles
201
202         energy_array[i_step] = total_energy
203
204         if np.mod(i_step + 1, freq) == 0:
205             # Update output file
206             traj.write(str(num_particles) + '\n\n')
207             for i_particle in range(num_particles):
208                 traj.write("%s %10.5f %10.5f %10.5f \n" % (
209                     element, coordinates[i_particle][0], coordinates[
    i_particle][1], coordinates[i_particle][2]))
210
211             # Adjust displacement
212             if tune_displacement:
213                 [n_trials, n_accept, max_displacement] = adjust_displacement(
214                     n_trials, n_accept, max_displacement)
215
216             # Print info
217             print(i_step + 1, energy_array[i_step])
218
219     traj.close()
220
221     end = time.time()
222     delta_t = end - start
223     print(f'Average total energy of particles of last 1000 steps: {np.mean(
    energy_array[-1000:])}' )
```

```
224     print(f'Time elasped: {delta_t} seconds')
225     print(
226         f'simulation speed: {delta_t / n_steps   1000} seconds per 1000 steps'
    )
227
228     rc('font',   { 'family': 'sans-serif',
229                    'sans-serif': ['DejaVu Sans'], 'size': 10})
230     # Set the font used for MathJax - more on this later
231     rc('mathtext',   { 'default': 'regular'})
232     plt.rc('font', family='serif')
233
234     plt.plot(range(5001, 5001 + len(energy_array[5000:])), energy_array
    [5000:])
235     if max(5001 + len(energy_array[5000:])) >= 10000:
236         plt.ticklabel_format(style='sci', axis='x', scilimits=(0, 0))
237     plt.xlabel('Monte Carlo steps')
238     plt.ylabel('Energy (reduced units)')
239     plt.title('Particle energy as a function of Monte Carlo steps')
240     plt.grid(True)
241     plt.savefig('Energy_plot.png')
242     plt.show()
243
244
245     plt.figure()
246     ax = plt.axes(projection='3d')
247     ax.plot3D(coordinates[:, 0], coordinates[:, 1], coordinates[:, 2], 'o')
248     ax.set_xlabel('X axis')
249     ax.set_ylabel('Y axis')
250     ax.set_zlabel('Z axis')
251     plt.title('The final configurations in 3D space')
252     plt.savefig('final_configuration.png')
253     plt.show()
```

Before we start modifying the code in the following chapter, we have to first realize the code. Here are some comments and explanantion about the code:

(1) Execution of the code
One key to a full understanding about this code is to realize how exactly the special variables `__name__` and `"__main__"` in Line 113 work.

- If we run the module (the source file) `MCLJ_original.py` as the main program, i.e. execute `python MCLJ_original.py`, the Python interpreter will assign the hard-coded string `"__main__"` to the `__name__` variable. That is, it's as if the interpreter inserts `__name__ = "__main__"` at the top of the module when we run as the main program.
- On the other hand, suppose this module is imported by another module, then in some other module the main program imports like this: `import MCLJ_original`. In this case, the interpreter will look at the filename of the module `MCLJ_original.py`, strip off the `.py`, and assign the remaining string to our module's `__name__` variable. That is, it's as if the interpreter inserts `__name__ = "MCLJ_original"` at the top of the module `MCLJ_original.py` when it is imported from another module.
- The advantage of putting actions to be executed after the statement `if __name__ == "__main__":` is that it separates the function defined in the module from the actions to be executed. That is, if we want to import the source code in another module, the action in `MCLJ_original.py`

will not be executed, since at this time we have `__name__ = "MCLJ_original"` instead of having `__name__ = "main"`. If we don't use these special variables, when we import the source code just to use the function defined in the module, the simulation will also be trigger, which is not what we want.

- Reference: https://stackoverflow.com/questions/419163/what-does-if-name-main-do

(2) Parameter setup and coordinate initialization

- From Line 117 to Line 128, the parameters are set to be the same as the ones adopted by NIST benchmark ($T* = 0.9$, $\rho* = 0.9$, and $r_c = 3\sigma$). On the other hand, from Line 130 to Line 140, to compare the result with NIST benchmark, we used 500 particles and set the reduced density as 0.9. What is noteworthy is that in this case, we generate the initial configuration randomly to initialize the simulation through the function `generate_initial_coordinates`. (As for the number of steps, we only used 1000000 steps instead of to save time.)
- The function `generate_initial_coordinates` (from Line 9 to Line 18) has four inputs, including `method`, `file_name`, `num_particles` and `box_length`. Note that all the arguments are keyword arguments (default arguments), which means that they are followed by an equal sign and an expression that gives its default value. Especially, setting the default of the arguments `file_name`, `num_particles` and `box_length` as `None` exempts our need to always specify a value for these arguments. For example, when we use the "random method", we don't have to input a coordinate file. If the argument `file_name` is instead a positional argument (non-default argument), which always requires value of the argument to be specified, then we will get an error like: `TypeError: generate_initial_coordinates() missing 1 required positional argument: 'file_name'`. On the other hand, if we set the default of the variable `file_name` as `None`, then the value of `file_name` has been specified, and we will not get the `TypeError` if we don't specify the file name when using the random method. Similarly, when we use the "file method", we will not get the `TypeError` when we don't specify the value of `num_particles`, `box_length`, which are already specified or can be calculated given the coordinate file. (Note: the positional arguments should always be put in front of the keyword arguments, if any.)
- In Line 13, we use `coordinates = (0.5 - np.random.rand(num_particles, 3)) * box_length` instead of `coordinates = np.random.rand(num_particles, 3) * box_length` to make sure that the average of the coordinates is approximately at the origin (0, 0, 0).

(3) Simulation initialization

- The section of simulation initialization ranges from Line 142 to Line 165, which calculates the initial value of the total pair energy and correction energy using functions `calculate_total_pair_energy` and `calculate_tail_correction`, which involves other functions like `lennard_jones_potential`, `minimum_image_distance`. Also, a file `traj.xyz` is created to be written down the trajectory of the configuration during the simulation.
- Each particle interacts with every other particle within the cutoff. Therefore, to calculate the system energy, we have to find out all the pairs and sum up their (pairwise) Lennard-Jones potentials. Note that in Line 45, `j_particle` loops over `i_particle` instead of `range(particle` to avoid double counting.

(4) Metropolis Monte Carlo algorithm

- The section of Metropolis Monte Carlo algorithm ranges from Line 167 to Line 219. In Line 179, the random displacement is defined as `random_displacement = (2.0 * np.random.rand(3) - 1.0) * max_displacement` so that it ranges from $[-\text{max\_displacement}, \text{max\_displacement}]$. Then, after the current energy, proposed energy, hence the energy difference are calculated by the func-

tion `get_particle_energy`, whether the move is accepted or rejected is decided by the function `accept_or_reject`. Accordingly, the energy is recorded in `energy_array` after the trial.

- Note that in the section of parameter setup, the frequency `freq` as set as 1000 steps. This is the frequency for the adjustment of the max displacement and writing out the energy value. Since `tun_adjustment` is set as `True`, every 1000 steps the if statement in Line 211 will always be triggered, which involves the function `adjust_displacement`. In the function, the acceptance rate will be calculated. Since `n_trial` will be set to zero after the if statement is executed, the denominator of the acceptance rate is always 1000 steps (frequency). If the acceptance rate is smaller than 0.38, then the maximum displacement will be adjusted to 80% of its original value to increase the acceptance. On the other hand, if the acceptance rate is larger than 0.42, the maximum displacement will be adjusted t0 120% of its original value. By doing so, we can ensure that the average acceptance rate is close to 40%, which is a reasonable value for Monte Carlo simulation. After all the Monte Carlo steps are executed, the trajectory file will be closed and saved (Line 219) and the average total particle energy of last 1000 steps will be printed out. In addition, the timer will stop and print out the time elapsed during the simulation, which can be used to assessed the efficiency of the code.

(5) Data visualization and the simulation result
As shown in Figure 3. and Figure 4., after all the Monte Carlo steps are finished, two plots will be generated. One is the particle energy as a function of Monte Carlo steps (starting from 5000-th steps). The other is the final configuration in 3D space. As a result of 1000000 steps, the total energy of 500 particles converges to XXXX. This value is pretty close to the NIST benchmark, which is -6.1773 obtained from a simulation with 5×10^7 steps equilibration and 2.5×10^8 steps of production. In addition, it turns out that it took XXX to finish 1000000 steps of simulation. That is, the simulation speed is xxxxxx seconds per 1000 steps. In the following chapter, this is one thing that we have to improve. Also note that in the following chapters, we will cover basic knowledge required to complete this project. While this documentation is project-oriented, in each section, we will begin with more general cases/examples to illustrate or implement important concepts, followed by parts that are more project-based.

Chapter 2: Package Installations and Virtual Environments

Completing the project requires some packages to be installed either using `pip` or `conda`. In addition, to separate the dependencies used for this project from those in the base environment, we have to create a virtual environment for this project. In this chapter, the basic knowledge about `pip` and `conda` as well as the creation of a virtual environment will be introduced. Also, in the last section, we will guide the reader to set up the system for the project.

2-1. Introduction to `pip`

(1) Definition from Wikipedia: `pip` is a de facto standard package-management system used to install and manage software packages written in Python. Many packages can be found in the default source for packages and their dependencies - Python Package Index (PyPI).

(2) Installation of pip
`pip` is already installed if one is using Python 2 later than version 2.7.9 or Python 3 later than version 3.4 downloaded from python.org or if one is working in a virtual environment created by `virtualenv` or `pyvenv`. About more information about `pip`, one can check out the documentation: https://pip.pypa.io/en/stable/

(3) Some common `pip` commands:

- To download a package, execute `pip download some-package-name`.
- To install a package, execute `pip install some-package-name`.
- To uninstall a package, execute `pip uninstall some-package-name`.
- To search for packages whose name or summary contains <query >, execute `pip search <query>`.
- To show information about one or more installed packages, execute `pip show some-package-name`.
- To check if all dependencies are compatible (missing packages/wrong versions), execute `pip check`.
- To list the packages that have been installed and their versions, execute `pip list`.
- To output installed packages in requirements format, execute `pip freeze`. This is useful to list all the package requirements of a software. To generate a requirements file `requirements.txt` and install from it in another environment, we can first execute `pip freeze > requirements.txt` and execute `pip install -r requirements.txt` in the target environment. If the version of a package is not specified in `requirements.txt`, `pip install -r requirements.txt` will install the newest version.

2-2. Introduction to virtual environments

(1) What is a virtual environment?
A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them. This is one of the most important tools that most of the Python developers use.

(2) Why do we need a virtual environment?
Imagine a scenario where we are working on two web based python projects and one of them uses a Django 1.9 and the other uses Django 1.10 and so on. In such situations virtual environment can be really useful to maintain dependencies of both the projects.

(3) When and where to use a virtual environment?

- By default, every project on our system will use these same directories to store and retrieve site packages (third party libraries). How does this matter? Now, in the above example of two projects, we have two versions of Django. This is a real problem for Python since it cannot differentiate between versions in the "site-packages" directory. So both v1.9 and v1.10 would reside in the same directory with the same name. This is where virtual environments come into play. To solve this problem, we just need to create two separate virtual environments for both the projects.The great thing about this is that there are no limits to the number of environments we can have since they're just directories containing a few scripts.
- Virtual Environment should be used whenever we work on any Python based project. It is generally good to have one new virtual environment for every Python based project we work on. So the dependencies of every project are isolated from the system and each other.

(4) To create isolated Python environments, we need the module `virtualenv`, which can be installed by executing `pip install virtualenv`. Then, to create a virtual environment named `test`, execute `virtualenv test`. This command creates a directory called `test`, which contains a directory structure similar to this (explanation provided in the parenthesis):

```
├── bin (files that interact with the virtual environment)
│   ├── activate (activate scripts are used to set up the shell to use the
│   │       environment's Python executable and its site-packages by default.)
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.7
│   ├── pip
│   ├── pip3
│   ├── pip3.7
│   ├── python
│   ├── python3 -> python
│   ├── python3.7 -> python
│   └── python-config
├── include (C headers that compile the Python packages)
│   └── python3.7m -> /home/wei-tse/anaconda3/include/python3.7m
│       (A lot of .h files)
└── lib (a copy of the Python version along with a site-packages folder where each
    │    dependency is installed)
    └── python3.7
        └── site-packages (a lot of .py files)
```

(5) To use the virtual environment `test`, execute `source test/bin/activate`. To exit the environment, use `deactivate`. One can use `which python` or `pip list` to check differences between environments.

(6) Here is a good documentation to look up more information about virtual environments:
https://realpython.com/python-virtual-environments-a-primer/

2-3. Management of virtual environments

(1) While virtual environments certainly solve some big problems with package management, they create some problems of their own after several environments are created, most of which revolve around managing the environments themselves. To help with this, the `virtualenvwrapper` tool was created. It's just some wrapper scripts around the main virtualenv tool. (To install, just execute `pip install virtualenvwrapper`.) A few of the more useful features of `virtualenvwrapper` are that it:

- Organizes all of our virtual environments in one location
- Provides methods to help us easily create, delete, and copy environments
- Provides a single command to switch between environments

(2) Once `virtualenvwrapper` is installed, we'll need to activate its shell functions. We can do this by running `source` on the installed `virtualenvwrapper.sh` script. When it is first installed by `pip`, the output of the installation show the exact location of `virtualenvwrapper.sh`. Or we can simply run `which virtualenvwrapper.sh`.

(3) First, we can make a directory `envs` for all the virtual environments (`mkdir envs` (directory of `envs`: /home/wei-tse/envs, or `~/envs` in short). Then, define the environment variable by running `export WORKON_HOME= ~/envs`. At last, use `which virtualenvwrapper.sh` to find out the path of `virtualenvwrapper.sh` and source it. After this, we can use commands like `mkvirtualenv`.

(4) To create a virtual environment named `test`, run `mkvirtualenv test`. The difference between `mkvirtualenv test` and `virtualenv test` is that the former always create the virtual environment in the folder `envs` (if the environment variable was defined correctly) and the command activates the environment to be created automatically. (Note: For environments created by either way, deleting the folder results in the deletion of the corresponding virtual environment.)

(5) To use the environment created by `mkvirtualenv`, in this case, `test`, execute `workon test`. Note that the environments created by `virturalenv` can not be accessed by the command `workon`. To exit the environment, use `deactivate`. Simply executing `workon` lists all the environments created by `mkvirtualenv`.

(6) To install all the packages based on the requirements file `requirements.txt` and a certain Python version (for example Python 3.7) and associate an existing project directory with the new environment to be created, we can use the following command: (The direcotry `python_path` can be found by using the command `which python`.)
`mkvirtualenv env_name -python=python_path -a project_path -r requirements.txt`.
Using this command, we can create virtual environments with differnt versions of Python so that we can switch Python versions quickly.

(7) Note that in the folder `bin` in the environment folder, there are shell scripts like `postactivate` and `preactivate`. (We can check the first few lines by using `more postactivate`.) Right now, both scripts are bascially empty, but we can add some commands so that these commands will be executed right after or before the environment is activated.)

(8) For more information, check: https://virtualenvwrapper.readthedocs.io/en/latest/

2-4. Introduction to `conda`

(1) Use of Anaconda with its package manager, `conda`, greatly simplifies package installation and environment management.

(2) `conda` is a general package manager, meaning that it can install dependencies and packages in languages besides Python, unlike `pip` (which is Python's package manager). Both `pip` and `conda` can be used to install packages.

(3) To create an environment using `conda`, execute `conda create -name env-name python=x.xx`.

(4) Use command `conda activate env-name` to activate an environment. (And use `condat deactivate` to deactivate.)

(5) Use command `conda info --envs` to see a list of all environments.

(6) Use command `conda list` to list all the Python packages installed in an environment and use command `conda install package-name` to install a package.

2-5. Project-based section: system setup for the MolSSI summer school project

To setup the system, readers can refer to the following website:
https://molssi-education.github.io/2019-software-summer-school-logistics/Setup.html or follow the instructions here (which is more simplified but basically the same).

- First, we use the command `conda create -n MCLJ_software python=3.7` to create a virtual environment called `MCLJ_software` with Python 3.7 installed.
- Then, install the following Python libraries:

```
conda install numpy
conda install matplotlib
conda install jupyter
conda install qcportal -c conda-forge
```

- In the following chapters, the reader will be required to install diffeent packages into this environment (`MCLJ_software`).

Chapter 3: Fundamentals of Git and GitHub

Whenever we collaborate on a project with others, either on this MolSSI project or any other project, it's a good practice to keep track of the version of the codes or packages under development. In this chapter, we will introduce one of the most commonly used software for version control - GitHub, which is also going to be used in the development of this project. Starting with different types of version control system (CVS), we will introduce the concepts and the importance of version control. Then, we will introduce some basic Git commands and the fundamentals of GitHub. In the end, we will recommend a common workflow that the reader can follow when collaborating with others on a common project, which should be also useful when developing the MolSSI project.

3-1. Introduction to version control system (CVS)

(1) Version control
A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision" (like revision 1, revision 2, ... etc).

(2) Version control system (VCS)
Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, collaborative web docs[2] and in various content management systems, e.g., Wikipedia's page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming in wikis. As shown in Figure 5. and Figure 6., version control system (VCS) can usually be divided into centralized version control system (CVCS) and distributed version control system (DVCS).

(3) The importance of version control
Reference: https://bit.ly/2YAFoos

- Collaboration
  - Without a VCS in place, we're probably working together in a shared folder on the same set of files. Shouting through the office that we are currently working on file "xyz" and that, meanwhile, our teammates should keep their fingers off is not an acceptable workflow. It's extremely error-prone as we're essentially doing open-heart surgery all the time: sooner or later, someone will overwrite someone else's changes.
  - With a VCS, everybody on the team is able to work absolutely freely - on any file at any time. The VCS will later allow you to merge all the changes into a common version. There's no question where the latest version of a file or the whole project is. It's in a common, central place: your version control system.
- Storing versions properly
  Saving a version of the project after making changes is an essential habit. But without a VCS, this becomes tedious and confusing very quickly:
  - How much should we save? Only the changed files or the complete project? In the first case, we'll have a hard time viewing the complete project at any point in time - in the latter case, we'll have huge amounts of unnecessary data lying on our hard drive.
  - How do we name these versions? If one is a very organized person, he/she might be able to stick to an actually comprehensible naming scheme (if he/she is happy with "acme-inc-redesign-2013-11-12-v23"). However, as soon as it comes to variants (say, we need to prepare one version with the header area and one without it), chances are good we'll eventually lose track.
  - The most important question, however, is probably this one: How do we know what exactly is different in these versions? Very few people actually take the time to carefully document each important change and include this in a README file in the project folder.

A version control system acknowledges that there is only one project. Therefore, there's only the one version on our disk that we're currently working on. Everything else - all the past versions and variants - are neatly packed up inside the VCS. When we need it, we can request any version at any time and we'll have a snapshot of the complete project right at hand.

- Restoring Previous Versions
  Being able to restore older versions of a file (or even the whole project) effectively means one thing: we can't mess up! If the changes we've made lately prove to be garbage, we can simply undo them in a few clicks. Knowing this should make you a lot more relaxed when working on important bits of a project.
- Understanding What Happened
  Every time we save a new version of your project, our VCS requires us to provide a short description of what was changed. Additionally (if it's a code / text file), we can see what exactly was changed in the file's content. This helps us understand how our project evolved between versions.
- Backup
  A side-effect of using a distributed VCS like Git is that it can act as a backup; every team member has a full-blown version of the project on his disk - including the project's complete history. Should our beloved central server break down (and our backup drives fail), all we need for recovery is one of our teammates' local Git repository.

(4) Centralized version control system (CVCS)

- A centralized version control system works on a client-server model. There is a single, (centralized) master copy of the code base, and pieces of the code that are being worked on are typically locked, (or "checked out") so that only one developer is allowed to work on that part of the code at any one time. Access to the code base and the locking is controlled by the server. When the developer checks their code back in, the lock is released so it's available for others to check out.
- Of course, an important part of any VCS is the ability to keep track of changes that are made to the code elements, and so when an element is checked in, a new version of that piece is created and logged. When everyone has finished working on their different pieces and it's time to make a new release, a new version of the application is created, which usually just means logging the version numbers of all the individual parts that go together to make that version of the application.
- When working with a centralized verison control system, our workflow for adding a new feature or fixing a bug in our project will usually look something like this:
  - Pull down any changes other programmers have made from the central server.
  - Make changes and make sure they work properly.
  - Commit the changes to the central server, so other programmers can see them.
- Examples include CVS, Subversion (SVN) and Perforce, ..., etc.

(5) Distributed version control system (DVCS)

- More recently, there's been a trend toward distributed version control systems. These systems work on a peer-to-peer model: the code base is distributed amongst the individual developers' computers. In fact, the entire history of the code is mirrored on each system.
- There is still a master copy of the code base, but it's kept on a client machine rather than a server. There is no locking of parts of the code; developers make changes in their local copy and then, once they're ready to integrate their changes into the master copy, they issue a request to the owner of the master copy to merge their changes into the master copy.
- With a DVCS, the emphasis switches from versions to changes, so a new version of the code is simply a combination of different sets of changes, which is quite a fundamental change in the way many developers work and is why a DVCS is sometimes considered harder to understand than a CVCS.
- Examples include Git, Mercurial, Bazaar, ..., etc.

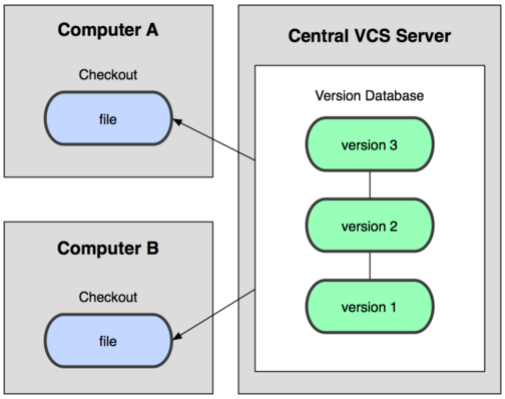

Figure 3: Centralized version control system

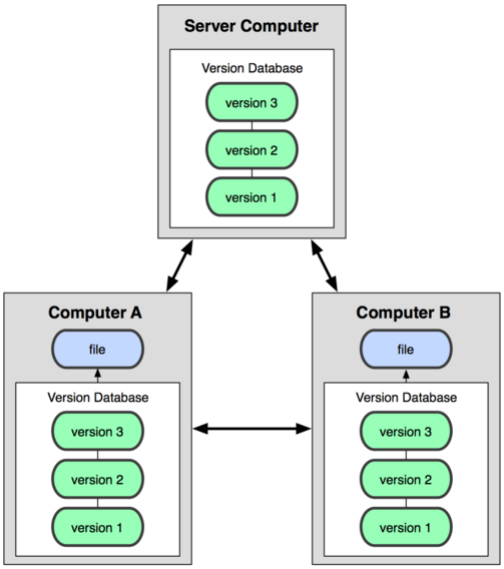

Figure 4: Distributed version control system

(6) Advantages of DVCS over CVCS
The act of cloning an entire repository gives distributed version control tools several advantages over centralized systems:

- Performing actions other than pushing and pulling changesets is extremely fast because the tool only needs to access the hard drive, not a remote server.
- Committing new changesets can be done locally without anyone else seeing them. Once one have a group of changesets ready, one can push all of them at once.
- Everything but pushing and pulling can be done without an internet connection. So one won't be forced to commit several bugfixes as one big changeset.
- Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback before showing the changes to everyone.

(7) Disadvantages of DVCS compared to CVCS
To be quite honest, there are almost no disadvantages to using a distributed version control system over a centralized one. Distributed systems do not prevent one from having a single "central" repository, they just provide more options on top of that. There are only two major inherent disadvantages to using a distributed system:

- If one's project contains many large, binary files that cannot be easily compressed, the space needed to store all versions of these files can accumulate quickly.
- If one's project has a very long history (50,000 changesets or more), downloading the entire history can take an impractical amount of time and disk space.

3-2. Basic commands of Git

(1) Setup and initialization of Git

- Download/install Git via https://git-scm.com/downloads
- To initialize, use the commands `git config --global user.name "user-name"` and `git config --global user.email email-address` to set up the ID and the email, respectively.

- Some certain Git commands will open text files. When this happens, Git will use the environment's default text editor, which might not be the editor that the user are most comfortable using. Using configuration commands, we can tell Git to use the preferred editor. For example, we can run `git config -global core.editor "emacs"` or `git config -global core.editor "vim"`.
- Use the command `git config --list`, we can check the configuration commands that we have set. (Or we can check the file `.gitconfig` in the home directory.)

(2) Basic commands of git

- Getting and creating projects

Command	Description
`git init`	Initialize a local Git repository
`git clone ssh://git@github.com/[usrname]/[repo-name].git`	Create a local copy of a remote repository

- Basic snapshotting

Command	Description
`git status`	Check status
`git add [file-name]`	Add a file to the staging area
`git add -A`	Add all new and changed files to the staging area
`git commit -m "[commit message]"`	Commit changes
`git rm -r [file-name]`	Remove a file (or folder)

- Branching and merging

Command	Description
`git branch`	List branches (the asterisk denotes the current branch)
`git branch -a`	List all branches (local and remote)
`git branch [branch name]`	Create a new branch
`git branch -d [branch name]`	Delete a branch
`git push origin --delete [branch name]`	Delete a branch
`git checkout -b [branch name]`	Create a new branch and switch to it

- Branching and merging (cont'd)

Command	Description
`git checkout -b [branch name] origin/[branch name]`	Create a remote branch and switch to it
`git checkout [branch name]`	Switch to a branch
`git checkout -`	Switch to the branch last checked out
`git checkout -- [file-name.txt]`	Discard changes to a file
`git merge [branch name]`	Merge a branch into the active branch
`git merge [source branch] [target branch]`	Merge a branch into a target branch
`git stash`	Stash changes in a dirty working directory
`git stash clear`	Remove all stashed entries

- Inspection and comparison

Command	Description
`git log`	View changes
`git log --summar`	View changes (detailed)
`git diff [source branch] [target branch]`	Preview changes before merging

- Sharing and updating projects

Command	Description
`git push origin [branch name]`	Push a branch to the remote repository
`git push -u origin [branch name]`	Push changes to remote repository (and remember the branch)
`git push`	Push changes to remote repository (remembered branch)
`git push origin -delete [branch name]`	Delete a remote branch

- Sharing and updating projects (cont'd)

Command	Description
`git pull`	Update local repository to the newest commit
`git pull origin [branch name]`	Pull changes from remote repository
`git remote add origin ssh://git@github.com/[usrname]/[repo-name].git`	Add a remote repository
`git remote set-url origin ssh://git@github.com/[usrname]/[repo-name].git`	Set a repository's origin branch to SSH

3.3 Project-based section: create a GitHub repository for the MolSSI project

In this section, we will demonstrate the use of the basic Git commands mentioned above in a project-oriented manner, along with some small experiments in attempt to provide a more clear demonstration.

(1) Basic snapshotting

- `git init`
  To begin with, we can first create a directory, say `MCLJ_software` for the project, enter the folder, and use the command `git init` to initialize this local Git repository. After the initialization, a folder `.git` will be created (can only be found by `ls -a`), which contains information about the repository (like repository branches or information about different version of the repository).
- `git status` Here we place the code for the MC simulation `MCLJ_original.py` in the folder `MCLJ_software`, which can be downloaded from https://github.com/wehs7661/MCLJ_software. Then, if use `git status` to check the status, we can see that the file `MCLJ_original.py` is in the section of "Untracked files", which means that it is still not under version control. (For more information about the tracked and untracked files, please refer to https://bit.ly/2JebXyH)
- `git add`
  To add `MCLJ_original.py` to the staging area, which is the area containing files ready to commit, execute `git add MCLJ_original.py`. If we check the status again, we should see that `MCLJ_original.py` is now in the section of "Changes to be committed". (For more information about stating area, please refer to https://bit.ly/2tyZIpd)
- To commit the file(s), use `git commit`:
  - If we execute `git commit` without specifying any filenames, then all the files in the staging area will be committed. (The files which haven't been added to the stating area by the command `git add` will not be committed.) After the command, we will be required to enter the commit message for the changes. (An empty message aborts the commit.)
  - Alternatively, we can directly use the command `git commit -m "This is the first commit of MCLJ_original.py" MCLJ_original.py`, we can commit only `MCLJ_original.py` instead all the files, with a commit message (though we have no other files right now).
  - After committing the file, we should see a message indicating the number of file changed (in this case, 1), number of insertions(+) and the number of deletions(-). The message followed `git status` shows `nothing to commit, working tree clean`. We can also go to the folder

`.git` and take a look at the file `COMMIT_EDITMSG`, which should show the commit message that we just entered. The default file name can be specified if we don't use the `-m` option and choose to edit in the prompt.

- `git log`
  Using `git log`, we can view the change that we just made, including the commit message. (Typically, we can see all the changes made recently.)
- `git diff`
  - To realize the command, now, we can add a line like `print "Hello Git!"` to the top of the file `MCLJ_original.py` and save the change. In the output of `git status`, it is shown that the file `MCLJ_original.py` was modified. Then, if we execute `git diff`, we should get a message indicating the difference the local version and the remote version of `MCLJ_original.py` in the repository, with the first line showing `print 'Hello Git!'`. To commit the change to the repository, we have to run `git add` and `git commit` again.
  - Since we added one line, after committing `MCLJ_original.py`, the message followed will indicate `1 file changed, 1 insertion(+)`.
  - This is just a experiment to demonstrate the command `git diff` and we don't need the line we just added in the project, so delete the line.

(2) Branching and Merging

- Prerequisite 1 of `git branch`: What is Git branching?
  - Unlike other VCS, the way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day.
  - To really understand the way Git does branching, we need to take a step back and examine how Git stores its data. Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots.
  - When we make a commit, Git stores a commit object that contains a pointer to the snapshot of the content we staged. This object also contains the author's name and email address, the message that we typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.
  - A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is `master`. As we start making commits, we're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.
  - The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the `git init` command creates it by default and most people don't bother to change it.
  - Check https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell for more information about Git branching.
- Prerequisite 2 of `git branch`: Why do we need Git branching?
  - Why do we need Git branching? Assume that in real world you are working on a web site following these steps:
    * Do work on a web site.
    * Create a branch for a new story you're working on.
    * Do some work in that branch.

  - At this stage, imagine you receive a call that another issue is critical and you need a hotfix. You'll have to do the following:
    * Switch back to your production branch.
    * Create a branch to add the hotfix.
    * After it's tested, merge the hotfix branch, and push to production.
    * Switch back to your original story and continue working.
  - For more details, refer to https://git-scm.com/book/en/v1/Git-Branching-Basic-Branching-and-Merging (Strongly recommended)
- `git branch` and `git checkout`
  - First, by running `git branch test`, we can create a branch named `test` (while still staying in the `master` branch). Or we can also use the command `git checkout -b test` to create the `test` branch and switch to it. In the `test` branch, there is also a file `MCLJ_original.py` exactly the same as the one in `MCLJ_original.py` branch.
  - Simply running `git branch` shows the list or all the branches and the branch where we are. In our case here, two branches, including `* master` and `test` will be shown and the branch where we are has a asterisk in front of the branch name. Run `git checkout test` to switch to the branch `test`.
  - To look into what Git branching really is, add one more line `print 'Here is master branch'` to `main.py` and switch to `test` branch.
  - Take a look at `main.py` in `test` branch, you should find that `main.py` is exactly the same as the one in `master` branch. That is because we did not add `main.py` to the stating area and commit it to the repository.
  - So now, go back to `master` branch, add/commit `main.py`, and take a look at `main.py` in `test`. You will find that there is no line like `print 'Here is master branch'` in `main.py`. That is because the change was committed to the repository in `master` branch and now the two `main.py` are different and separate from each other.
  - Add a line `Here is test branch` to `main.py` in `test` branch. If we now want to use `git checkout master` to switch back to `master` branch, we will encounter an error like `your local changes to the following files would be overwritten by merge:main.py`. That is, after the first change is committed to the repository, whenever we change the file(s) in either branches, we have to add the changed file to the staging area and commit it before we switch to other branches.
- `git merge`
  - We can use `git merge [branch name]` to merge a branch into the active branch.
  - However, occassionly `git merge` doesn't go smoothly. If we changed the same part of the same file differently in the two branches we're merging together, Git won't be able to merge them cleanly, which is actually case rigt not. (The line `Here is master branch` and `Here is test branch` appears at the same line.) That is, if we directly execute `git merge` here, we'll get a merge conflict that looks something like this:

```
Auto-merging main.py
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

  - At this stage, Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. To see which files are unmerged at any point after a merge conflict, we can run `git status`.

– Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
print 'Hello Git branch!'

<<<<<<< HEAD
print 'Here is test branch'
=======
print 'Here is master branch'
>>>>>>> master
```

– Run `git branch -d [branch name]` to delete the branch if you feel necessary after merging.

3-4. Fundamentals of GitHub

(1) GitHub is an American company that provides hosting for software development version control using Git. To begin with, go to the GitHub website (https://github.coms/) to register an account.

(2) `git clone`

To create a local copy of a remote repository, we can use the command `git clone [http:// ...]`. Useing `git log` and `git branch`, we can also check the changes that have been made and what branches does the repository contain. Note that if the user has not set the ID and the email using `git config`, the commit message shown by `git log` will show that the repository has an "invalid email address.

(3) `git push`

Conversely, if we want to push the local repository, say, `udemy_test` in our case, to GitHub, there are several ways to do it:

- First log in GitHub, click on the plus sign on the right upper corner of the page to create a new repository named `udemy_test` and decide if the repository should be public or private. Since we are not create an empty repository, but copy our local repository to GitHub, which has already been initialized locally. Therefore, here we don't have to initialize the repository. After clicking on "Create repository", we should see the instruction page as shown below.
- As shown in the Figure 4. in the next page, to create an exactly same repository `udemy_test` on GitHub, we can either upload existing files or push an existing repository from the command line. In our case, run `git remote add origin https://github.com/wehs7661/udemy_test.git` and `git push -u`
  `origin master` and we will be required to enter the username and the password of the GitHub account on the terminal. Finally, after reloading the GitHub page, we should see that all the files contained in `udemy_test` are uploaded.
- If we makes changes locally after pushing the repository the GitHub and we want the changes to be updated on GitHub, we have to add the changed files to the stating area, commit them to the repository and execute `git push origin master` to push the updated repository to GitHub.
- Note that right now the repository on GitHub only have one branch `master`, since we only pushed one branch just now. To push the other branch `test`, we only have to run the push command again: `git push origin test`. Even if we create other branches after pushing the repository, this command still works.
- A refernce for one FAQ (remote origin already exists while pushing): https://tinyurl.com/y3xzvm87

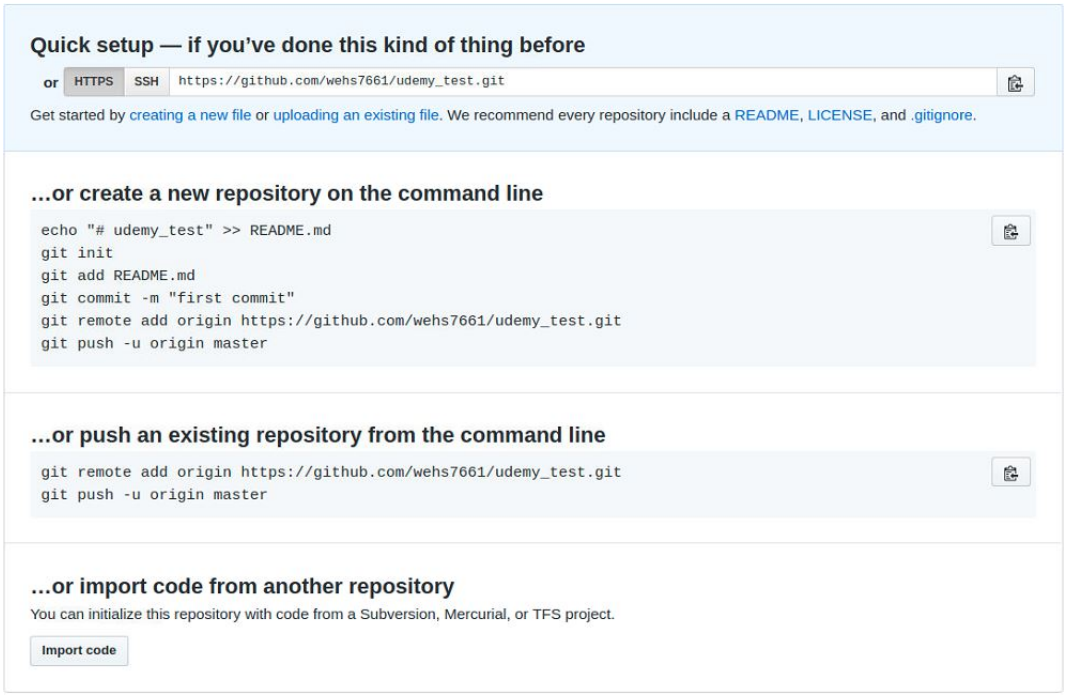

Figure 5: Instructions on repository creation on GitHub

(4) SSH key for GitHub authentication
As one might easily find, we have to enter the username and the password of the GitHub account whenever we use `git push` (or other commands that require the access to the GitHub account). To make the validation process easier and actually safer, we can generate an SSH key for the GitHub authentication. To set up an SSH key, we basically follow the instructions on https://tinyurl.com/y59795nn and https://tinyurl.com/y5qbbra7 as follows:

- In the terminal, we first execute `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`.
- Then, in the prompts followed, just press `Enter` three times, to accept the default file location and no passphrase. After that, the key will be generated. The prompt and the result are like the following

```
Enter file in which to save the key (/home/wei-tse/.ssh/id_rsa):
/home/wei-tse/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/wei-tse/.ssh/id_rsa.
Your public key has been saved in /home/wei-tse/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:vOjxtTC+hu++LCvI04D6T3Gu4z/6raVZG8pua/RbZY4 wehs7661@colorado.
    edu
The key's randomart image is:
+---[RSA 4096]----+
|                 |
|                 |
|                 |
|       .         |
| .   . . S   o   |
```

```
|. .   +.. . =      |
|.. +..+oB E .      |
|. +.++%oB .        |
| ..+=X&@&+.        |
+----[SHA256]-----+
```

- Then, we should copy the content of the public key (the file `id_rsa.pub`, whose location should be shown in the message obtained in the process of generating the key) to GitHub. To do this, first turn to the GitHub page, click on the "Settings" button listed after the click on the account icon on the right upper corner of the page. In the personal settings list, click on "SSH and GPG keys" and click on "New SSH key" on the right.
- On the right of the page, we decide the title (for example, "personal laptop" or "Lab desktop" and paste the content of `id_rsa.pub` to "Key" column. Subsequently, click on "Add SSH key" and confirm the GitHub password.
- To be able to use `git push` without entering username and password, we have to first execute `ssh -T git@github.com`. Then, check the remote origin by running the command `git remote -v`. If the origin is shown as follows:

```
origin    http://github.com/wehs7661/udemy_test.git (fetch)
origin    http://github.com/wehs7661/udemy_test.git (push)
```

  then we have to use the following command to `http` to `ssh`:

  `git remote set-url origin git@github.com:wehs7661/udemy_test.git`

  After that, if the get the following result after running `git remote -v` again, we should be able successfully use the SSH key when using `git push` or other commands which require access to GitHub.

```
origin    git@github.com:wehs7661/udemy_test.git (fetch)
origin    git@github.com:wehs7661/udemy_test.git (push)
```

  (Basically, with `git remote set-url` is doing is to modify the URL of the remote origin in the file `config` (in the folder `.git`) from `http://github.com/wehs7661/udemy_test.git` to `git@github.com:wehs7661/udemy_test.git`.)

(5) Contribute to others projects

- If we want to make changes to others projects. We can "fork" the repository that we want to change. After using `git clone`, we are able to modify the files in the repository, add, commit and push to our own GitHub account. (Note that the changes we made on the forked repository will not be updated to the original repository.)
- If we want to make changes to the original repository, on the GitHub page, we can send "pull request" (click on "New pull request" on the page of the original repository). Then the page will show the difference between the files in our forked repository and the files in the original repository. After we leave message about the change has been made on the page and "Create pull request", the page of the original repository will show a new button "pull request" in the bar at the top list of the page.
- Once our pull request is created, the project owner can review the request and decide if he or she want to merge the pull request. This is how we can contribute to others repositories.
- One of the common ways to contribute to others project is to do code review on GitHub, as introduced in this YouTube video: https://www.youtube.com/watch?v=HW0RPaJqm4g

(6) Frequently asked questions about forking and cloning a repository

- What is the difference between forking and cloning a repository?
  - Forking
    A fork is a copy of a repository that allows us to freely experiment with changes without affecting the original project. A forked repository differs from a clone in that a connection exists between your fork and the original repository itself. In this way, the fork acts as a bridge between the original repository and our personal copy where we can contribute back to the original project using Pull Requests.
  - Cloning
    * When we create a new repository on GitHub, it exists as a remote location where our project is stored. We can clone a repository to create a local copy on our computer so that we can sync between both the local and remote locations of the project.
    * Unlike forking, we won't be able to pull down changes from the original repository you cloned from, and if the project is owned by someone else you won't be able to contribute back to it unless you are specifically invited as a collaborator. Cloning is ideal for instances when you need a way to quickly get your own copy of a repository where you may not be contributing to the original project.
  - Will the fork contain the same data as the original project?
    Forking a repository will copy the main data such as files and code. Issues, branches, pull requests and other features, however, will not copy over to your fork. Instead your fork will start the same way as a newly created repository, but with all of the content present at the time of forking, so you can work on it as a fresh project.
  -
  - When should I fork a repository?
    If we want a link to exist between our copy of a project and the original repository, we should create a fork. This will allow us to make changes to ourr fork, then open a pull request to the original to propose your changes. Forking is ideal for open-source collaboration, as it allows for anyone to propose changes to a project that the original repository maintainer can choose to integrate.

  - If I want to back up my repository, should I clone it?
    Cloning a repository is a great way to create a backup. However, while our clone will copy over Git data like files and commit history, it won't bring over issues, pull requests, and other GitHub elements.
  - When should I clone a repository?
    If we require a copy of a project, and do not need to sync your copy with the original project? If so, a clone is more suitable because it does not share a connection with the original repository.

  To clone a repository, head over to the main page of a project and click the Clone or download button to get the the repository's HTTPS or SSH URL. Then, qw can perform the clone using the 'git clone' command in your command line interface of choice.

3-4. Project-base section: common GitHub workflow for collaboration on the project

Chapter 4: Object-Oriented Programming (OOP) and Design Patterns

4-1. Objects in Python programming

4-2. Four Pillars of object-oriented programming

4-3. Design Pattern 1: Decorator

4-4. Design Pattern 2: Abstract factory

4-5. Design Pattern 3: Setter and getter

4-6. Project-based section: code refactoring by design patterns

Chapter 5: Introduction to C++ and C++ binding

5-1. Data type in C++

5-2. Basic coding of C++

5-3. Binding C++ to Python

5-4. Project-based section: code acceleration using C++ binder

Chapter 6: Parallel Computing in Python

6-1. Introduction to supercomputers and HPC systems

6-2. Parallel computing with MPI

6-3. Parallel computing with OpenMP

6-4. Project-based section: code acceleration using parallel computing

Chapter 7: Unit Testing and Continuous Integration

7-1. Packing for unit testing and estimation of code coverage

7-2. Project-based section: testing codes for the Monte Carlo simulation

7-3. Travis CI and Codecov

Chapter 8: Python Coding Style and Documentation

8-1. Introduction to PEP8 coding style

8-2. Useful Python formatter

8-3. Style of the docstring

8-4. Python documentation generator: `Sphinx`

8-5. Publishing and editing of the documentation

8-6. Create, edit, and publish a documentation using `mkdocs`