Many Particle Systems

Simulation and Modeling (CSCI 3010U)

Faisal Qureshi

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Many particle systems

We now turn our attention to *many particle systems.* These simulations lie at the intersection of deterministic and random simulations.

These simulations are often used to model solids, liquids and gases. The idea is to gain insight into micro structure of these materials by simulating individual particles. It is also possible to study the macro structure, which leads us to thermodynamics and random processes.

Molecular dynamics

- Gases, liquids and solids are composed of molecules.
- Even a small sample of material can contain a *very* large number of molecules: on the order of 10^{25}.
- It is possible to study these systems by simulating molecules. We obviously cannot yet simulate 10^{25} molecules. We can, however, study thousands or even millions of molecules.
- We will use *small scale* simulations (say 100 or 1000 particles) learn how these materials behave.
- We will make simplifying assumptions about how molecules behave (interact, move, etc.). It turns out that individual motion of each molecule is not important. Rather we are interested in the statistical properties of the entire collection of molecules.

Molecular dynamics — key assumptions

- Shape is not important
- There are no chemical reactions
- Molecules follow classical dynamics
- Gravity is ignored
- Inter-molecular forces (attraction and repulsion) are important
  - These forces depend only upon the distance between molecules

Molecular dynamics — potential energy

Each molecule pair contribute $u(r_{ij})$ to the total *potential energy* of the system. r_{ij} is the distance between molecules i and j.

$$U = \sum_{j=1}^{N-1} \sum_{j=i+1}^{N} u(r_{ij})$$

We can compute U using quantum dynamics, however, that is too difficult. Rather we will construct phenomenological models based upon observations, and use these models to estimate U.

Lennard-Jones potential

- Molecules attract each other due to *van der Walls* interactions (if the molecules are not too far apart)
- Molecules repulse each other when they get too close to each other (Pauli exclusions principle)
  - This is akin to collisions between particles

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

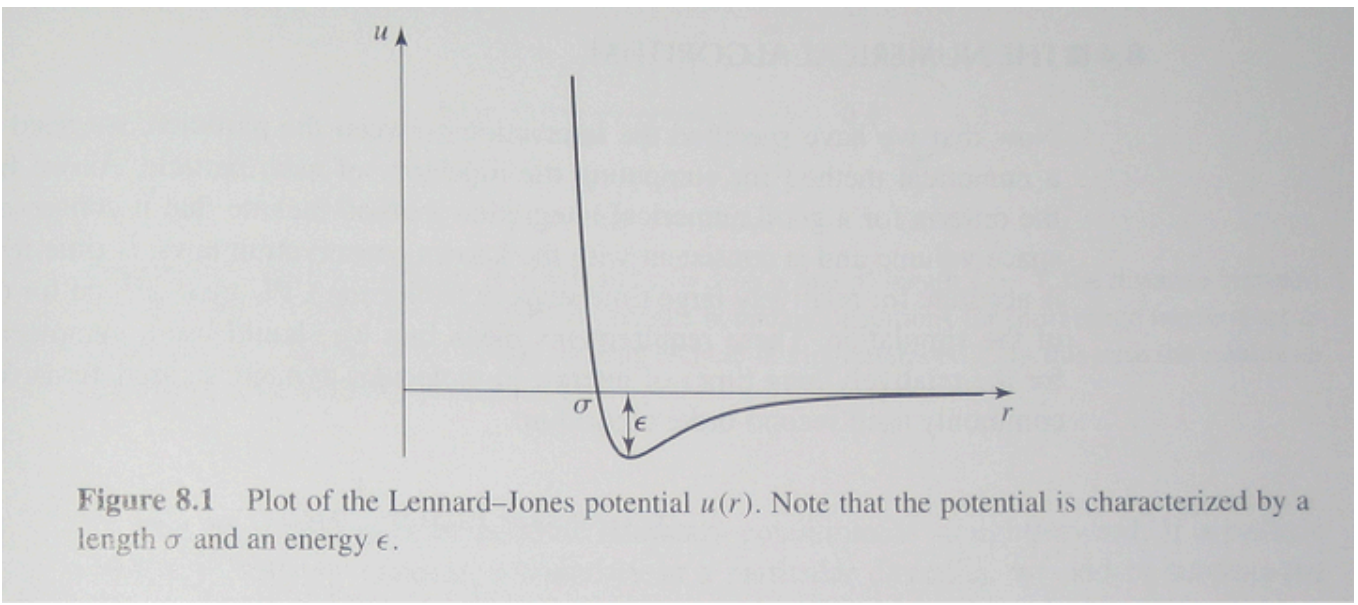


Figure 8.1 Plot of the Lennard–Jones potential $u(r)$. Note that the potential is characterized by a length σ and an energy ϵ.

Molecular dynamics — implementation considerations

Many of the quantities used in molecular dynamics are quite small. This can lead to numerical errors (loss of precision, problem with divisions, etc.)

An option is to scale these quantities.

Lennard-Jones potential gives us two natural units for our simulations.

- Use σ as the unit of distance.
- Use ϵ as the unity of energy.
- For mass, we use the mass of a single atom.
- Velocity is measured in units of $(\epsilon/m)^{1/2}$
- Time is measured in units of $\sigma(\epsilon/m)^{1/2}$
  - Our unit of time is 2.17×10^{-12} seconds
- Molecular dynamics simulations typically run for 10^{-11} to 10^{-8} seconds

Physical units for Argon gas

Quantity	Unit	Value for Argon
length	σ	3.4×10^{-10} m
energy	ϵ	1.65×10^{-21} J
mass	m	6.69×10^{-26} kg
time	$\sigma (m/\epsilon)^{1/2}$	2.17×10^{-12} s
velocity	$(\epsilon/m)^{1/2}$	1.57×10^{2} m/s
force	ϵ/σ	4.85×10^{-12} N
pressure	ϵ/σ^2	1.43×10^{-2} N $\cdot$ m^{-1}
temperature	ϵ/k	120 K

Computing accelerations

In order to simulate particle motion, we need quantities for the standard equations of motion: force, mass and acceleration.

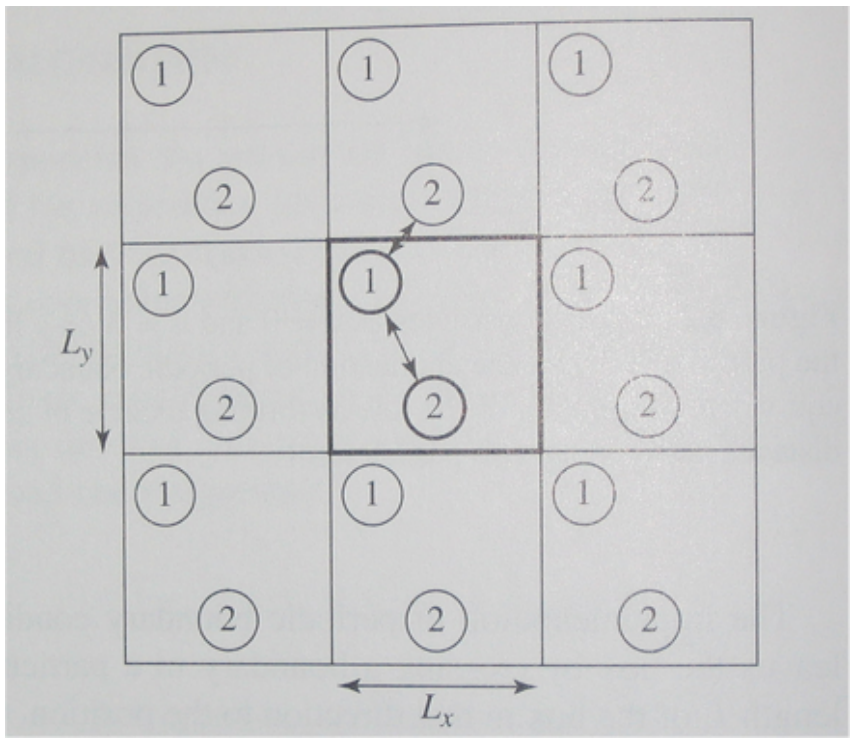
We can get the force (and thereby acceleration) from the potential energy computed via Lennard-Jones potential.

$$\begin{aligned} f(r) &= -\nabla u(r) \\ &= \frac{24\epsilon}{r}\left[2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \end{aligned}$$

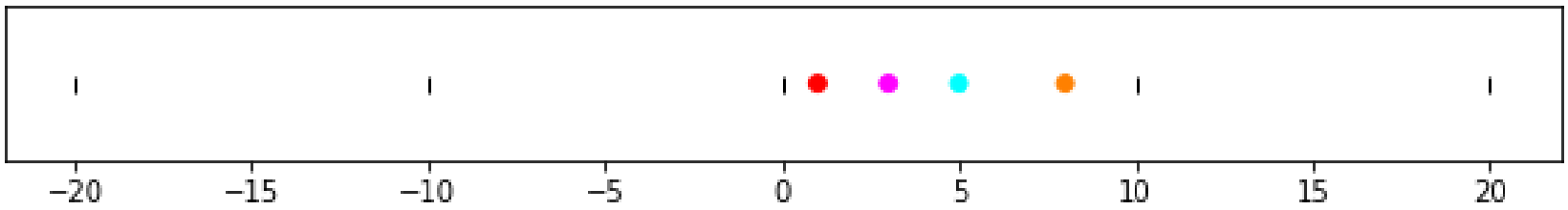
Boundary conditions

When we have a very large number of molecules (say 10^{25}) in a very small space, only a tiny fraction of these molecules interact with the boundaries.

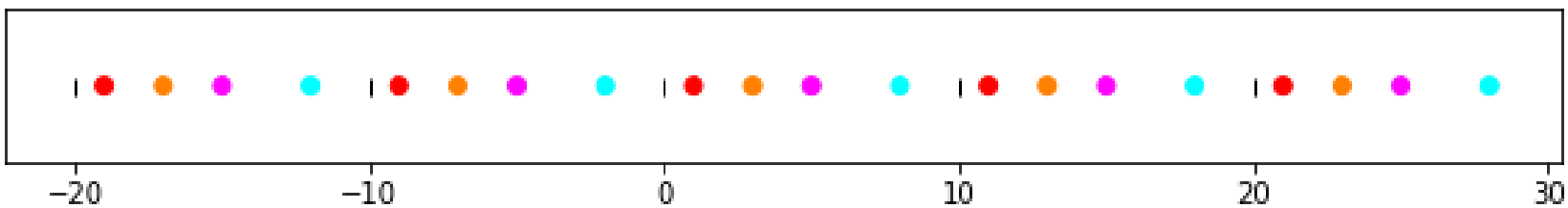
In our simulations, we are simulating far fewer particles. Consequently, a large fraction of these end up interacting with the boundaries. We need to pay special attention to boundary conditions in our simulations. Periodic boundary conditions are often used that remove the surface effects from the simulation.



Boundary conditions - positions



Positions



Positions with periodic boundary conditions ($L_x = 10$)

Boundary conditions - position

```
def pbc_pos(p, l):
    if p > 0:
        while p > l:
            p -= l
    elif p < 0:
        while p < 0:
            p += l
    else:
        pass
    return p
```

Example

Positions: $1, 8, 13, -5$

Positions with periodic conditions ($L_x = 10$): $1, 8, 3, 5$

Boundary conditions - separation

```
def pbc_sep(d, l):
    if d > 0:
        while d > 0.5*l:
            d -= l
    elif d < 0:
        while d < -0.5*l:
            d += l
    else:
        pass
    return d
```

Example

Positions: $1, 8, 13, -5$
Distances (from 1): $0, -2, -4, -7$

Distances with periodic conditions ($L_x = 10$): $0, -2, -4, 3$

Setting up initial conditions

One of the challenges in these simulations is how to set up the initial conditions.

We are often interested in the overall state of the system (say, as described by some differential equation, some statistical property of the system, etc.). How do we then set up the individual particles (say their locations and velocities) such that the overall state of the system is what we desire?

Setting up the locations on a rectangular grid

```
def rect_pos(nx, ny, Lx, Ly):
    '''
    Create 4 x n_particles state vector
    '''
    n_particles = nx * ny                # number of particles
    state = np.zeros([4, n_particles])   # 4-by-n_particles state vector
    dx, dy = Lx / nx,  Ly / ny           # the velocities are currently
    for ix in range(nx):                 # set to 0
        for iy in range(ny):
            i = ix + iy * nx
            state[0,i] = dx * (ix + 0.5) # x location of ith particle
            state[1,i] = dy * (iy + 0.5) # y location of ith particle
    return state
```

Initial conditions example - system temperature

According to the equipartition theorem, the mean kinetic energy of a particle per degree of freedom is $kT/2$, where k is Boltzmann's constant and T is the temperature. We can generalize this relation to define the temperature at time t by

$$kT(t) = (2/d)K(t)/N,$$

where K is the kinetic energy of the system at time t and d is the spatial dimension. In the following we will consider $d = 2$.

How do we setup individual velocities to have certain temperature $T(t)$ at time $t = 0$?

Kinetic energy

Recall that kinetic energy at time t is

$$\frac{1}{2}\sum_{i=1}^{N} m\mathbf{v}_i \cdot \mathbf{v}_i$$

Setting up initial velocities - getting the momentum right

Before we begin to assign velocities (and sometimes locations) to our particles, we make the following observations: *the whole material is not in motion, so the overall momentum must be 0*

1. Randomly assign velocities to particles
2. Compute overall momentum of the system
3. Compute average momentum (per particle)
4. Subtract average momentum (per particle) from each particle

Now the total momentum of the system is 0

Setting up initial velocities - getting the momentum right

```
v_sum = np.zeros([1,2])
n_particles = state.shape[1]

for i in range(n_particles):
    state[2:, i] = np.random.rand(2) - 0.5
v_sum = np.sum(state[2:, :], 1)

for i in range(n_particles):
    state[2:, i] = state[2:, i] - (v_sum / n_particles)

# Now the net velocity of all particles is [0,0]
```

Setting up initial velocities - getting the kinetic energy right

Now we turn our attention to temperature of the system. Temperature of the system is a function of kinetic energy. We compute the total kinetic energy of the system, and use it to compute a scale factor that will take the current kinetic energy of the system to the desired kinetic energy.

Multiply all velocities with this scale factor to achieve the desired kinetic energy.

Setting up initial velocities - getting the kinetic energy right

```
v2_sum = np.sum(state[2:,:]**2, 1)

total_ke = 0.5 * (v2_sum[0]+v2_sum[1]) # 1/2 m v^2, assuming m = 1
ke_per_particle = total_ke / n_particles
rescale_ke = (initial_ke_per_particle / ke_per_particle) ** 0.5
state[2:, :] = state[2:, :] * rescale_ke
```

Setting up intitial conditions - getting the locations right

Now that we have the right velocities for our particles. How do we assign positions to these particles. Of course no two particles should occupy the same position.

- For gases, place particles at random positions
- For solids, place particles at lattice locations
- For liquids, particles locations fall the two other alternatives

Setting particles positions for gas simulations

- Place particles at random locations
- Particles cannot be too close to each other
  - This will generate very large repulsive forces, which in turn will cause problems for our ODE solvers
  - Separation between two molecules should be at least $2^{1/6}\sigma$
- Use rejection based techniques to achieve this
  - Generate a location, and accept it if it is not too close to an existing molecule
  - This approach works reasonably well as long as gas is not very dense

Setting particle positions for gas simulations

One option to deal with dense gases is to start with a solid. Place particles at grid locations. Then slowly rise the temperature of the system. Solid will turn to liquid, which in time will turn to gas.

This also solves the problem of setting particle positions for liquids.

This approach raises an interesting question. Often time initial conditions will be *unstable* and these will slowly evolve towards *equilibrium*.

Setting up initial conditions

- Setting up initial conditions can take a long time
- It is useful to have the ability to save and load initial conditions

Practical considerations

- Many simulations run for hours if not days and weeks. The ability to save/load the state of simulation is important. If something fails, the simulation can be restarted.
  - These are called *checkpoint dumps* or simply *checkpoints*
  - Many supercomputing centers forces you to save simulations periodically.

Simulation particle dynamics

Now that the initial conditions are set — each particle is at a particle location, moving with a certain velocity — we will use equations of motions as before to evolve the simulation over time i.e., particle locations and velocities as they move under the influence of attractive and repulsive forces.

We need to solve $F = ma$

- $m = 1$ in our units. So the above equation is simplified to $F = a$.
- We need to compute accelerations produced by each molecule pair.
  - Each molecule belongs to $N - 1$ pairs
- Note that if molecule i exerts force f_{ij} on molecule j then molecule j exerts $-f_{ij}$ on molecule i. So $f_{ij} = -f_{ji}$

Computing forces (accelerations)

```
def compute_acc(state, Lx, Ly):
    n_particles = state.shape[1]

    acc = np.zeros([2, n_particles])

    pe, virial = 0, 0
    for i in range(0, n_particles-1):
        for j in range(i+1, n_particles):
            dx = state[0,i]-state[0,j]
            dy = state[1,i]-state[1,j]
            dx = pbc_sep(dx, Lx)
            dy = pbc_sep(dy, Ly)
            r2 = dx**2 + dy**2
            one_over_r2 = 1. / r2
            one_over_r6 = one_over_r2**3
            f_over_r = 48. * one_over_r6 * (one_over_r6 - 0.5) * one_over_r2
            fx = f_over_r * dx
            fy = f_over_r * dy
            acc[:,i] = acc[:,i] + [fx, fy]
            acc[:,j] = acc[:,j] - [fx, fy]
    return acc
```

Verlet solver

- We can choose to use Verlet ODE solver for this application
- Notice that acceleration is computed twice at each step; however, if we are careful we only need to compute accelerations once

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_n \Delta t + \frac{1}{2} a_n (\delta t)^2$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \frac{1}{2} \left(a_{n+1} + a_n \right) \Delta t$$

Using simulations to carry out experiments

We are not interested in the positions/velocities for individual particles. Rather we are interested in the properties of the matter as a whole. We want to relate the simulation results to quantities that we can measure in the real world.

Instantaneous kinetic temperature

Lets consider *temperature*. We can measure instantaneous kinetic temperature using the following relationship

$$kT(t) = \frac{1}{2N} \sum_{i=1}^{N} m_i \mathbf{v}_i(t) \cdot \mathbf{v}_i(t).$$

Measuring kinetic temperature

The temperature that is usually measured in a molecular dynamics simulation is the time average of $T(t)$ over many configurations of the particles. We will refer to this temperature as the kinetic temperature. For two dimensions we write the kinetic temperature as

$$kT = \frac{1}{2N} \sum_{i=1}^{N} m \overline{\mathbf{v}_i(t) \cdot \mathbf{v}_i(t)}.$$

The bar denotes the time average. This equation only holds if the momentum of the center of mass is fixed.

Measuring pressure

With in the context of thermodynamics, the other quantity that is of interest is *pressure*

$$P(t)V = NkT(t) + \frac{1}{2}\sum_{i<j} \mathbf{r}_{ij}(t) \cdot F_{ij}(t),$$

where V is the volume (or area in 2D).

We cannot observe instantaneous pressure in the real world. So we instead we will use the time average

$$\frac{PV}{NkT} - 1 = \frac{1}{2NkT}\sum_{i<j} \overline{\mathbf{r}_{ij}(t) \cdot F_{ij}(t)}.$$

Using simulations to carry out experiments

Simulations can also be used to compute other real-world quantities of interest:

- heat capacity
- velocity distributions that describe the probability of a molecule having a particular velocity
- distributions of inter-molecule distances

Many particle simulations so far

- These simulations have started with completely deterministic motion at the molecular level, and from that we have produced statistics that link to results obtained in the real world.

Ensemble

Ensemble is a collection of states at the micro level that leads to the same results at the macro level.

For example, the set of all possible particle velocities that give rise to the same overall temperature.

Microcanonical ensemble

Assumption: no external forces (influences) on the particles, i.e., no gravity.

Consider N particles in a volume V with total energy E. The macrostate of this system is given by N, V and E. In this case many microstates satisfy a macrostate. Each of these microstates is often referred to as *accessible state*. We have no preference for any microstate. Consequently, each microstate is equally likely.

Say in total there are Ω microstate. Then the probability P_s of a particular microstate s is

$$P_s = \frac{1}{\Omega}$$

Why microcanonical ensemble?

Previously we were performing *time averages* to compute a macro value. For example, we need to average instantaneous temperatures over a short duration to get the temperature of the system. This is at least how a thermometer works in the real world.

In order to perform time averages we had to simulate system dynamics (using equations of motions).

Say there is a mechanism to enumerate any microstate s, along with its probability P_s. Additionally, that we are able to compute the macro property A_s given this microstate. We can then compute average of the macrostate as follows:

$$\langle A \rangle = \sum_{s=1}^{Q} P_s A_S$$

Quasi-ergodic hypothesis

It is sometimes more efficient to replace time average with $\langle A \rangle$. This is called quasi-ergodic hypothesis, which has been proved for many interesting systems (but not in general).

Why is this a big deal?

This means that we are able to compute time averages of (macro) quantities of interest without running time-consuming dynamics.

We do not need to compute system dynamics. We simply need to sample the states. The sampling has to unbiased, otherwise the average is not valid. Even if there is a very large number of states, we only need to sample a finite subset of those, since our time averages are also computed over a finite length of time.

How do we sample the states of an ensemble with the desired energy?

- We could just randomly generate the properties of the particles, but its unlikely that their total energy would be E
- We would have to generate a large number of states in order to get a small number that meet our criteria
- This is not very efficient, so we need a better way

Demon algorithm

- The demon algorithm is a technique that can be used to generate states that have a total energy of E
  - We do this by adding an extra degree of freedom called the demon
  - The demon can loan energy to the system or absorb extra energy to keep the total energy at E
  - We don't need to generate a set of particles with total energy exactly E

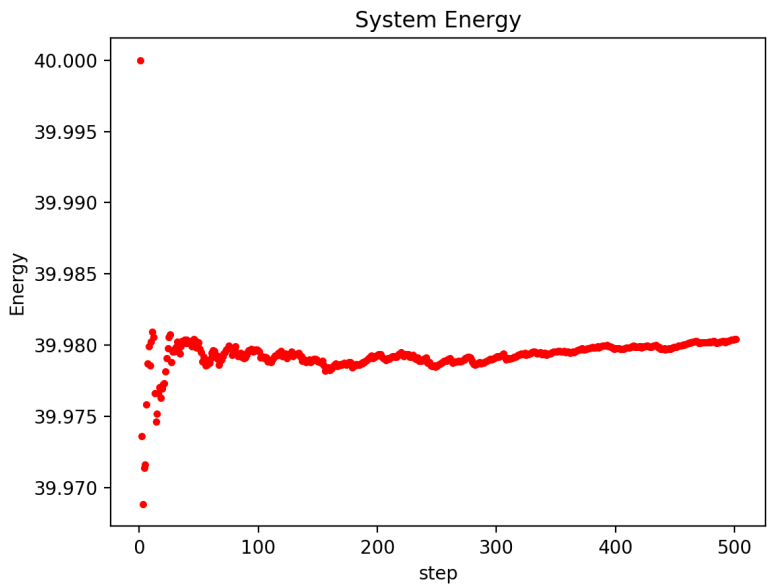
Demon algorithm

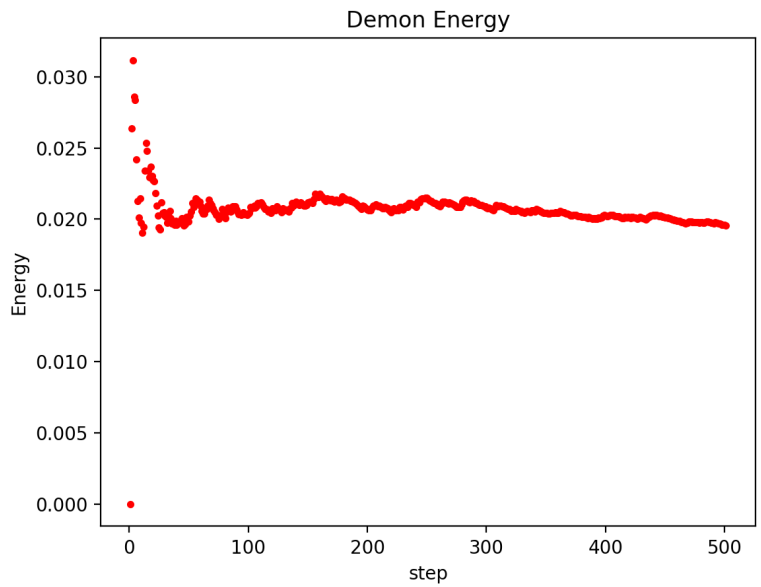
Choose a particle at random and make a trial change to its properties

- Compute ΔE, the change in energy due to this change
- If $\Delta E \leq 0$, accept the change and give $|\Delta E|$ to the demon, $E_d = E_d + |\Delta E|$
- If $\Delta E > 0$ and $E_d \geq \Delta E$, accept the change and remove energy from demon, $E_d = E_d - \Delta E$
- Otherwise, reject the change
- Repeat until equilibrium is reached, that is the changes in E_s (the energy of the system) and E_d settle down
  - Note that $E_s + E_d = E$ is a constant since energy flows between the demon and the system to keep the energy constant

Using demon algorithm to sample system states at energy 40 with 4000 particles

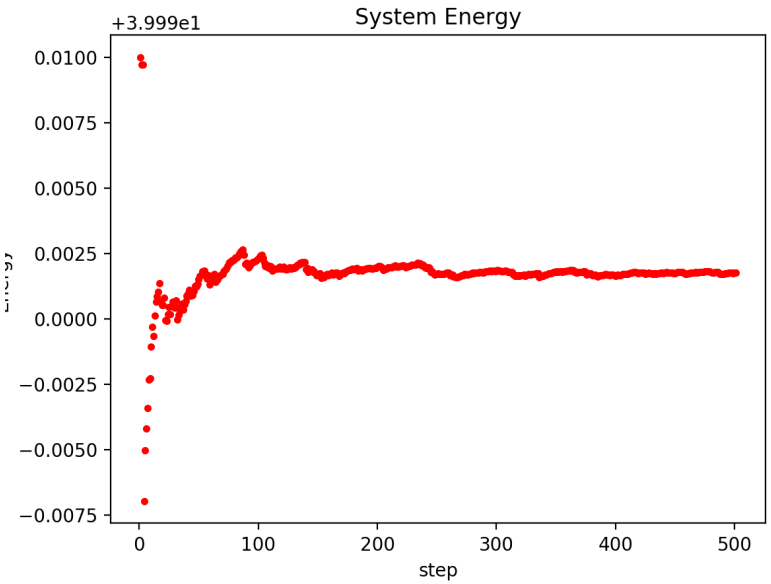
- After 500 steps the system and the demon reach a steady state where system energy hovers around 39.980
- $\langle E_d \rangle = 40.06$ and $\langle E_d \rangle = 0.02$

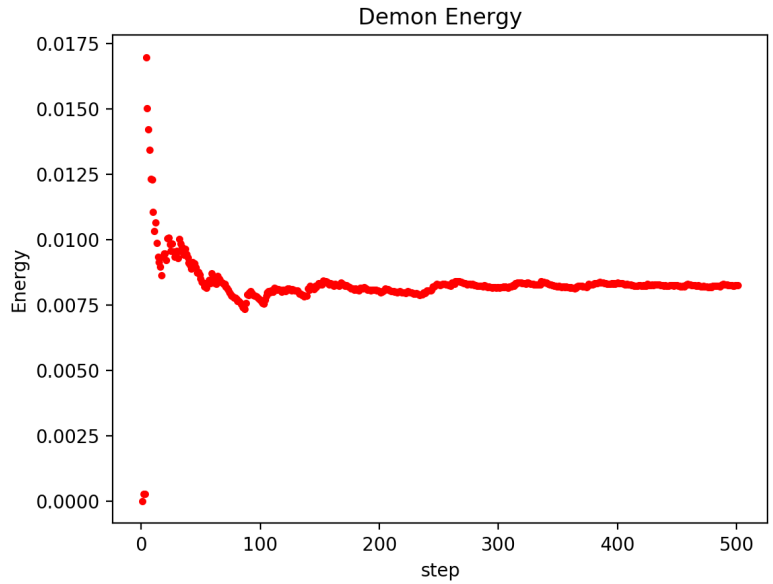




Using demon algorithm to sample system states at energy 40 with 10000 particles

- We can further reduce the effect of demon by using a larger N (for this experiment $N = 10000$)
- $\langle E_d \rangle = 40.07$ and $\langle E_d \rangle = 0.0083$





Demon algorithm

```
def mcs(N, demon_energy, system_energy, v, delta):
    accepted = 0
    for i in range(N):
        j = np.random.choice(N)
        old_v_j = v[j]
        change = (np.random.random()-0.5)*delta
        v[j] += change
        new_energy = compute_energy(v)
        delta_energy = new_energy - system_energy

        if delta_energy < 0:
            system_energy += delta_energy
            demon_energy -= delta_energy
            accepted += 1
        elif delta_energy > 0:
            if demon_energy >= delta_energy:
                system_energy += delta_energy
                demon_energy -= delta_energy
                accepted += 1
            else:
                v[j] = old_v_j

    return accepted, demon_energy, system_energy, v
```

Demon algorithm

- It turns out the demon itself tells us a lot of interesting information
- If we run some experiments we find out that the probability distribution of E_d is

$$P(E_d) \propto e^{-E_d/kT}$$

- We can use $P(E_d)$ to estimate the kinetic temperature of the demon. So demon can act as a thermometer.

Canonical ensemble

- The ensemble that describes the probability distribution of a system in thermal equilibrium with a heat bath is known as the *canonical ensemble*.
- A canonical ensemble is characterized by temperature T, the number of particles N and the volume V.
- We can treat demon to be the system of interest which is interacting with a much bigger system (an ideal gas in our example), which acts as the heat bath.
- The probability distribution of the microstates of a system in thermal equilibrium with a heat bath is the same as the probability distribution of the energy of the demon

$$P_s = \frac{1}{Z} e^{-\beta E_s},$$

where $\beta = 1/kT$ and Z is the normalisation constant. P_s is referred to as the Boltzmann or the canonical distribution. Z is the partition function.

Ising model

- A model for molecular behavior
- Proposed by Wilhelm Lenz 1920 and developed by his student Ernst Ising 1925 to study the phase transition from a paramagnet to ferromagnet
  - Paramagnetism is a form of magnetism whereby certain materials are weakly attracted by an externally applied magnetic field,
  - Ferromagnetism is the mechanism by which certain materials (such as iron) form permanent magnets, or are attracted to magnets.
- 2D and 3D Ising model exhibit phase transition

Ising model

- The lattice has N sites, and at each site i the molecule can be in one of two states s_i, where s_i can be either $+1$ or -1
- These states are called *spins*, since the original application was magnetic material
- The energy of the system is given by:

$$E = -J \sum_{i,j=nn(i)}^{N} s_i s_j - B \sum_{i=1}^{N} s_i$$

- The first term models the interaction between neighbouring molecules and the second models the effect of an external magnetic field
- If $J > 0$, the model favours pairs of molecules with the same spin
- If $J < 0$, the model favours pairs of molecules with the opposite spin
- Normally periodic boundary conditions are used with this model

Ising model and magnetism

- Magnetism is given by

$$M = \sum_{i=1}^{N} s_i$$

- Other quantities of interest $m = \frac{M}{N}$, the magnetism per spin, or $\langle M \rangle$ or $\langle M^2 \rangle - \langle M \rangle^2$
- Notice that this system doesn't have traditional dynamics, so we will sample its states using some type of Monte Carlo algorithm

Ising model and demon algorithm

- Use demon algorithm to sample configurations of the Ising model
  - Choose a spin at random, and flip it.
  - Compute the change in energy and decide to accept the change or not.
- Measure the temperature of the system by using demon energy E_d
  - Compute $P(E_d) \propto e^{-E_d/kT}$ and use it to find T
  - Estimate $\langle E_d \rangle$
- Note that $\langle E_d \rangle$ is not equal to kT since E_d in case of the Ising model is discrete. Rather E_d is related to temperature as follows

$$kT/J = \frac{4}{\ln(1 + 4J/\langle E_d \rangle)}$$

- It is difficult to pick an initial confguration (of spins) with precisely the desired energy
  - It is convenient to choose the initial energy of the system plus the demon to be an integer multiple of $4J$
  - Begin with an initial configuration where all spins are up and then randomly flip spins while the energy is less than the desired energy

Metropolis algorithm

- When a system is placed in thermal contact with a heat bath at temprature T, the system reaches thermal equilibrium by exchanging energy with the heat bath. Imagine a large number of copies of system with volume V, number of particles N at thermal equilibrium at temperature T then the probability P_s that the system is in microstate s with energy E_s is

$$P_s = \frac{1}{Z} e^{-\beta E_s}.$$

- We can use P_s to obtain ensemble average of the physical quantities of interest. E.g., the mean energy is

$$\langle E \rangle = \sum_s E_s P_s = \frac{1}{Z} \sum_s E_s e^{-\beta E_s}.$$

Metropolis algorithm

- Given m samples of the total number of M microstates, we an estimate the mean value of a physical quantity A as follows

$$\langle A \rangle \approx A_m = \frac{\sum_{s=1}^{M} A_s e^{-\beta E_s}}{\sum_{s=1}^{M} e^{-\beta E_s}},$$

  A_s is the value of the physical quantity in microstate s.
- A crude Monte Carlo will generated a microstate s at random, calculate E_s, A_s and $e^{-\beta E_s}$.
- A microstate thus generated would be very improbable and will contribute very little to the sum.
  - We need to be smarter in how we sample states.
  - Be careful that we don't introduce bias.

Metropolis algorithm - Importance sampling

- To introduce importance sampling lets rewrite

$$\langle A \rangle \approx A_m = \frac{\sum_{s=1}^{M} A_s e^{-\beta E_s}}{\sum_{s=1}^{M} e^{-\beta E_s}},$$

by multiplying and dividing by π_s:

$$\langle A \rangle \approx A_m = \frac{\sum_{s=1}^{M} A_s e^{-\beta E_s} \pi_s}{\sum_{s=1}^{M} e^{-\beta E_s} \pi_s}.$$

- If we generate microstates with probability π_s, then the above can be re-written as:

$$A_m = \frac{\sum_{s=1}^{M} (A_s/\pi_s) e^{-\beta E_s}}{\sum_{s=1}^{M} (1/\pi_s) e^{-\beta E_s}}.$$

- If we average over a biased sample generated according to π_s, we need to weight each microstate by $1/\pi_s$ to get rid of bias.

Metropolis algorithm - Importance sampling

- Generate microstate s with probability π_s, then

$$A_m = \frac{\sum_{s=1}^{M} (A_s/\pi_s) e^{-\beta E_s}}{\sum_{s=1}^{M} (1/\pi_s) e^{-\beta E_s}}.$$

- A reasonable choice for π_s is Boltzmann distribution

$$\pi_s = \frac{e^{-\beta E_s}}{\sum_{s=1}^{m} e^{-\beta E_s}}.$$

- If each microstate s is sampled according to the Boltzmann distribution then

$$A_m = \frac{1}{m} \sum_{s=1}^{m} A_s.$$

Metropolis algorithm for Ising model

1. Establish an initial microstate (the energy of this state is not important)
2. Choose a spin at random and make a trial flip
3. Compute $\Delta E = E_{\text{trial}} - E_{\text{old}}$.
4. If $\Delta E \leq 0$, accept the new microstate s.
5. If $\Delta E > 0$, compute $w = e^{-\beta \Delta E}$
6. Generate uniform random number r in unit interval $[0, 1]$
7. If $r \leq w$, accept the microstate, otherwise retain the old microstate
8. Determine the value of the desired physical quantity A_s
9. Repeat steps 2 through 8 to obtain a sufficient number of microstates
10. Periodically compute averages over the microstates

Getting rid of autocorrelations

- We do not want to compute A_s after each flip, because the the values of A_s before and after the flip will be very similar.
- Ideally we wish to compute A_s for states that are statistically independent. The problem is that we don't know *a priori* the mean number of spin flips that are needed to obtain configurations that are statistically independent.
- Use time displaced *autocorrelation* function

$$C_A(t) = \frac{\langle A(t_0)A(t_0+t)\rangle - \langle A\rangle^2}{\langle A^2\rangle - \langle A\rangle^2}$$

to find time interval t such that $A(t_0)$ and $A(t_0+t)$ become uncorrelated or in other words $C_A(t)$ becomes 0.

  - $C_A(t) \to 0$ as $t \to \infty$
  - $C_A(t=0)$ is normalized to unity.
  - For Ising model $t = 1$ refers to a single spin flip.

Autocorrelation

- We can re-write the time displaced autocorrelation function as follows:

$$C_A(t) = \frac{\frac{1}{N-t}\sum_{t_0=1}^{N-t} A(t_0)A(t_0+t) - \langle A\rangle^2}{\langle A^2\rangle - \langle A\rangle^2}.$$

- The denominator is there to normalize $C_A(0)$.

Summary

- Simulating gases, liquids and solids
- Molecular dynamics – Lennard-Jones potential
- Boundary conditions
- Microcanonical and canonical ensemble
- Demon algorithm
- Ising model
- Metropolis algorithm
- Autocorrelations