CHE384T PS1: Random Walk Diffusion

September 9, 2024

1 PS1: Random Walk Model for Diffusion

Pre-reqs: - jupyterlab-myst: https://github.com/executablebooks/jupyterlab-myst

1.1 Context and Motivations

In this problem set, we explore one of the simplest models for random walk diffusion. While the lattice models here are somewhat removed from an actual material, the random walk diffusion model will introduce many basic concepts behind computer simulations in materials science.

If atoms are restricted to lattice sites and all sites are occupied, there is no diffusion in a material. However, at any finite temperature temperature, there is a finite equilibrium concentration of vacancies. Atoms can then move through the lattice by jumping to an adjacent unoccupied lattice site.

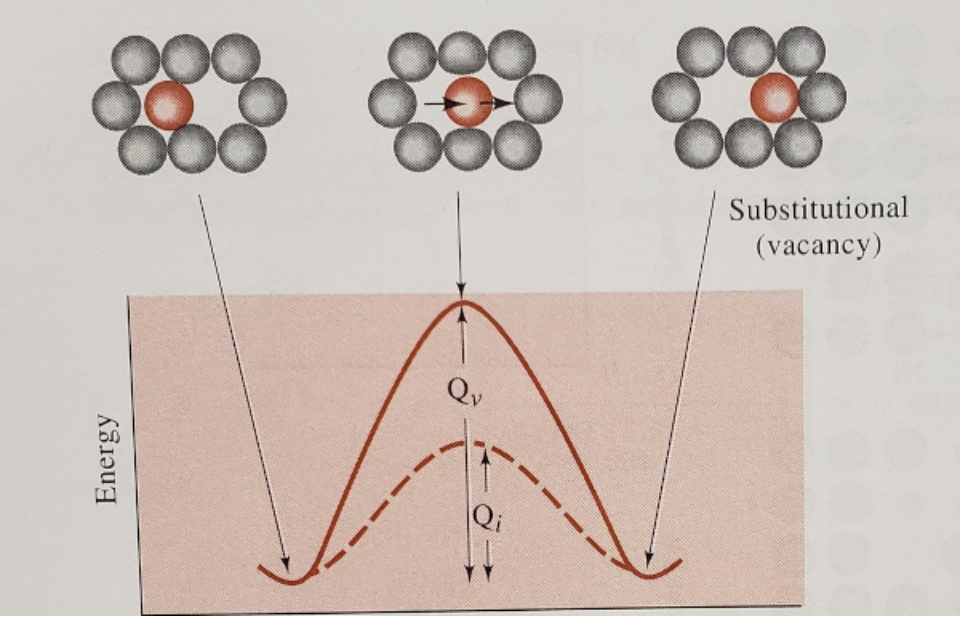
The simplest model for diffusion would then be to consider a single vacancy and its nearby atoms, measuring the vacancy diffusion constant. This low-concentration limit is called tracer diffusion.

In this exercise, we will construct at simple model for tracer diffusion. The diffusivity of atom may be described as an Arrhenius relation

$D = D_o \exp\left(-\frac{E_a}{k_B T}\right)$

That is, the diffusion of an atom to an adjacent lattice site must overcome an energetic potential barrier E_a. As is evident in the Arrhenius relation, the diffusivity increases with temperature, as atoms have more thermal energy to overcome the energy barrier.

A physical model to understand where this energy barrier comes from is shown in the figure below.



Consider how we might construct a simple model for tracer diffusion. As an atom jumps from its site to an adjacent occupied site, it must overcome a potential barrier, which depends on details of the local crystallography and temperature. To calculate an absolute diffusion rate requires then a determination of the activated jump process.

How can we model this? One could imagine using a method in which we calculate all the forces between atoms and solve the equation of motion of the atoms. This approach, called molecular dynamics, would give us accurate diffusion constants, depending on the quality of the models used to describe the interactions. We will cover the basics of molecular dynamics in a later problem set. Another approach, called Kinetic Monte Carlo, also incorporates details of interatomic interactions, but requires a great deal of development before we can apply it.

Instead, we may turn to a simpler model to extract essential characteristics of diffusion. The simplest of such models is a random walk model, in which most of the details are ignored: there are no interatomic interactions included in the model, the jump rates are assumed to be the same for all possible jumps and the timescale is measured relative to the jump rate, etc.

We start with a two-dimensional square lattice with one vacancy. One could do a simulation in which we try to move all the atoms. Since only those atoms next to a vacancy can move, it is equivalent to just move the vacancy. We will do a random walk, by which we mean we shall let it move around the system, where hops to any of the four nearest neighbors is chosen by a random number. We will then measure the mean square displacement $\langle r^2 \rangle$, which is related to the diffusion constant.

$D = \frac{1}{6t}\langle r^2 \rangle$

Here we outline the basic approach for a random walk on a two-dimensional square lattice. While we assume a square lattice, it is important to note that in this approach, there is actually no lattice. The symmetry is defined by the jumps. Thus, as we will find in following exercises, extension to other crystal systems is straightforward.

In this class, we will use Python, an open-source object-oriented programming language that is

widely used in the materials science community. Adaptation to other scientific programming languages such as Fortran, C++, or MATLAB would be relatively straight forward.

1.2 Random walk on a square lattice

[1]:
```
## Code for generating a single trajectory on square lattice

## a magic command that will transplant the *.py script directly as code
# %load lattice_2D.py

## a magic command that will show the contents of the *.py script
##   copy the code below into a file called lattice_2D.py in a folder called
 ↪code_exercises
##   and reload this cell.
%pycat lattice_2D.py
```

```
# Random walk on a 2D lattice

import numpy as np

def random_walk(nt,latt_type):
    """
    Random walk on a 2D lattice
    Inputs:
        nt [integer] = number of desired jumps (i.e., time steps)
        latt_type [string] = lattice geometry, current options
                             are 'square' and 'triangle
    Outputs:
        rs2 [array] = square displacement at each time step
        x   [array] = x-coordinate at each time step
        y   [array] = y-coordinate at each time step
    """
    # array for x- and y-coordinates along hopping path
    x = np.zeros(nt+1)
    y = np.zeros(nt+1)
    rs2 = np.zeros(nt+1)

    # particle starts at origin
    x[0] = 0
    y[0] = 0
    rs2[0] = 0

    ## square lattice
    if latt_type == 'square':
        # create a list of random numbers from 1 to 4 with nt entries
        fd = np.floor(4 * np.random.rand(nt))
        # next two lines define the jumps on the square lattice:
        #   right, up, left, down
```

```
        delx = np.array([1, -1, 0, 0])
        dely = np.array([0, 0, 1, -1])
    else:
        raise ValueError("Lattice type not implemented! See random_walk.py")

    # loop over nt jumps, add the jump vector as generated randomly in fd
    #sum over nt jumps
    for j in range(nt):
        x[j+1] = x[j] + delx[int(fd[j])] # x position at j+1 jump
        y[j+1] = y[j] + dely[int(fd[j])] # y position at j+1 jump

        # square displacement position at j+1 jump in 2D
        rs2[j+1] = x[j+1]**2 + y[j+1]**2

    return rs2, x, y
```

The code is relatively straightforward. - We create a set of arrays that will separately track the x- and y- coordinates - We initialize the position of the random walker to be at the origin for the first step - A list of nt randomly-generated numbers between 1 and 4 is initialized and will be used to choose the sequence of hops - We use the in-built `numpy` function `random.rand` - `random.rand` returns a list of random numbers between 0 and 1 (excluding 1) - Each entry is multiplied by the number of nearest neighbors to return an array of number between 0 and 4 (excluding 4) - In order to obtain of a list of randomly-generated integers as indices for choosing each hop, we use the `floor(x)` function, which rounds x down to the smaller integer.

For a square lattice, there are four possible jumps, given by four vectors:

```
delx(0), dely(0) = (1,0) = right
delx(1), dely(1) = (-1,0) = left
delx(2), dely(2) = (0,1) = up
delx(3), dely(3) = (0,-1) = down
```

In the `for` loop, we use the randomly-generated numbers in `fd` to index `delx` and `dely` in order to choose each subsequent hop, thus picking a random jump direction. The resulting displacement described by `delx` and `dely` is added to the current position. After each jump, the square displacement `rs2` is computed. The user determines the total number of steps taken in each trajectory by specifying `nt`. The function defined for random walk on a square lattice returns the square of the displacement `rs2` and the positions (x, y) at each step.

The function `random_walk_square` computes one trajectory (i.e., a sequence of jumps). In order to compute a mean square displacement , we need to run many trajectories and average over them.

[2]:
```
## Code for determining average mean square displacement over many trajectories

##   copy the code below into a file called avg_rand_walk2D.py in a folder␣
 ↪called code_exercises
##   and reload this cell.
%pycat avg_rand_walk_2D.py
```

```
import numpy as np
```

```
from lattice_2D import random_walk

def avg_rand_walk(nt, nd, latt_type):
    """ Average over many trajectories of random walkers
        Input:
            nt (integer) = number of jumps
            nd (integer) = number of trials
            latt_type (string) = type of lattice,
                                 "square" or "triangular" implemented
        Output:
            rwa (list; float) = mean square displacement of each trajectory
            ree (list; float) = end-to-end distance for each trial
            sig (list; float) = relative standard deviation of <r^2>
                                of each trajectory
    """

    # Initialize the variables for calculating <R^2> and <R^4>
    rwa = np.zeros(nt+1)
    ree = np.zeros(nd)
    sig = np.zeros(nt+1)

    # Loop over trials
    for j in range(nd):
        # Call random walker code from above to generate a trajectory
        g, x, y = random_walk(nt,latt_type)

        # Calculate end-to-end distance, 2D
        # index last element
        ree[j] = np.sqrt(x[-1]**2 + y[-1]**2 )

        # Increment R^2 and R^4 at each time step (jump)
        for k in range(nt+1):
            rwa[k] += g[k]
            sig[k] += g[k] ** 2

    # Find averages by dividing by the number of trials
    # Note: rwa[0] is 0 by definition, will throw a NaN warning
    for k in range(nt+1):
        rwa[k] /= nd
        sig[k] = (sig[k] / float(nd) - rwa[k] ** 2) / rwa[k] ** 2

    return rwa, ree, sig
```

1.3 ========== Exercises ==========

1.3.1 Random walk on different types of lattices

In these exercises, there is no prescription on how to organize your code, but it is highly recommended that you think about how to organize your code and anticipate modifications of new additions to the code.

1. Consider a random walk on a 2D square lattice:

    a. Use the code provided or write a new one and run it.

    b. Plot at least two trajectories and compare.

    c. Calculate the mean square displacement. Examine the behavior of the mean square displacement for a single run and for many runs.

    d. How many runs are needed to obtain a straight line for the mean square displacement?

2. Modify the code such that diffusion occurs on a 2D triangular lattice. Repeat the questions in 1. How do the results compare?

3. Modify the code such that diffusion occurs on a 3D simple cubic lattice. Repeat the questions in 1. How do the results compare?

4. Modify the code such that diffusion occurs on a 3D face-centered cubic lattice. Repeat the questions in 1. How do the results compare?

1.3.2 Statistics of the random-walk model

Consider 1D diffusion along the x-axis starting at $x = 0$ in step sizes of a. There is an equal probability the particle will hop to the left or right. After n jumps, we will record a final position x_n. Suppose we generate many such trajectories and average over them to find the probability of the particle at x_n along the 1D lattice. The probability of finding the particle at x_n would have the functional form of a Gaussian distribution.

$I(x_n) = \left(\frac{3}{2\pi n a^2}\right)^{1/2} \exp\left(-\frac{3x_n^2}{2na^2}\right)$

In three-dimensions, the corresponding probability distribution of the position is given by

$\mathcal{P}(\mathbf{R}_n) = I(x_n)I(y_n)I(z_n)$

where $\mathbf{R}_n = (x_n, y_n, z_n)$ and $I(y_n)$ and $I(z_n)$ have expressions similar to $I(x_n)$.

A more useful quantitify would be the probability distribution of the end-to-end distance $R_n = |\mathbf{R}_n|$, i.e., a measure of how far the atom particle has diffused in n steps. To find $\mathcal{P}(R_n)$, the angular information contained in the distribution of $\mathbf{R}_n$ vectors must be averaged out, which can be accomplished by tranforming to spherical polar coordinates and integrating out the angles. Once that is done, we find the probability distribution of the end-to-end distance in 3D is

$$\mathcal{P}(R_n) = \left(\frac{3}{2\pi n a^2}\right)^{3/2} 4\pi R_n^2 \exp\left(-\frac{3R_n^2}{2na^2}\right) \tag{1}$$

5. Make separate plots showing $I(x_n)$ and $\mathcal{P}(R_n)$. Provide a physical interpretation of the plot of $I(x_n)$ at large negative $-x_n$ values. How does the most probable point for $I(x_n)$ compare with that of $\mathcal{P}(R_n)$?

Supposed we have m random-walk simulations giving us m values for R_n after n jump sequences. Create a discrete representation of $\mathcal{P}(R_n)$ for a specific set of values of R_n by dividing the m values into user-defined bins that represent a finite-range of R_n. In other words, create histogram by breaking up the data into n_{bin} equally-spaced bins with bin width

$$\Delta = \frac{R_n^{max} - R_n^{min}}{n_{bin}} \tag{2}$$

Then determine m_i, number of R_n that lies within each ith bin. The probability of a value R_n is thus

$$\mathcal{P}_i = m_i/m \tag{3}$$

6. Run a series of trajectories and implement a binning procedure to calculate the probability distribution of the end-to-end distance across a series of trajectories. Compare your histogram of R_n from your computed trajectories with the predicted form of $\mathcal{P}(R_n)$.
   a. how many runs are needed to obtain well-converged results?
   b. how many bins are needed to obtain fine enough resolution?