CHE384T PS2: Lattice Sums, part 1

August 28, 2024

1 PS2: Lattice Sums and Simulation of finite systems

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

1.1 Context and Motivations

Methods to model materials typically involve computing a system with discrete objects, such as atoms, spins, or defects. The calculation itself may involve computing a total energy, interaction energy, order parameters, etc. Usually, we are faced with the challenge of modeling a macroscopic system with many components or objects, which would be computationally intractable to do in whole. Thus, we often use cutoffs and boundary conditions to approximate (effectively) finite systems.

As an illustration of the basic components of simulating (effectively) finite systems, we will perform lattice sums (of energy) on a crystalline lattice. We will describe how to set up the simulation of a cell that is replicated in space with periodic boundary conditions, which is particularly useful when describing materials with translation symmetry (e.g., crystals).

In materials modeling, summing over interactions is a common task. In this problem set, we will demonstrate the calculation of summing the interaction energy.

Consider a system of N atoms in a simulation cell in which the total interaction energy may be written as

$$U = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\phi_{ij}(r_{ij}) \tag{1}$$

where $i = j$ terms are omitted and ϕ_{ij} is a pair potential that depends only on the distance between atoms

$$r_{ij} = [(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2]^{\frac{1}{2}} \tag{2}$$

As N increases, so does the computation cost to perform the lattice sum. Thus, it is common practice to assume that at some separation, the interatomic potential is small enough to be negligible. The distance at which the interatomic potential is considered negligible is the cutoff distance, r_c. Note that slowly decaying potentials that go as $1/r^n$ where $n \leq 3$ require more sophisticated methods beyond using a simple cutoff distance.

The most straightforward method to include a cutoff is to explicitly test the distance between two atoms and not include interactions for $r_{ij} > r_c$. The task is then to sum over only the neighbors that are within the cutoff distance.

Many systems of interest are crystalline, meaning the material has translational symmetry and can be described using a unit cell with periodic boundary conditions. A central simulation cell is chosen and effectively replicated to fill space. The possible symmetries of a unit cell are described with the Bravais lattices. In this exercise, we will assume a cubic unit cell.

Consider an atom i at position $\vec{r}_i$. In a periodic cell, the replicas of atom i are located at $\vec{R} + \vec{r}_i$ where $\vec{R}$ is a lattice vector of the simulation cell. The energy then becomes

$$U = \frac{1}{2} \sum_{\vec{R}} \sum_{i=1}^{N} \sum_{j=1}^{N} \phi_{ij}(|\vec{R} + \vec{r}_j - \vec{r}_i|) \tag{3}$$

where we sum over the lattice vectors $\vec{R}$ and omit terms relating to $i = j$ and $\vec{R} = 0$. We will use periodic boundary conditions to mimic the properties of the larger (crystalline) system.

1.2 Implementation of Lattice Sums

1.2.1 Generating an FCC lattice

Let us generate a simulation over an FCC unit cell in the conventional cubic Bravais lattice. In the conventional FCC unit cell, there are four atoms with position vectors (written in fractional coordinates in terms of the lattice parameter a)

$$\begin{aligned} \vec{r}_1 &= (0, 0, 0) \\ \vec{r}_2 &= (\frac{1}{2}, \frac{1}{2}, 0) \\ \vec{r}_3 &= (\frac{1}{2}, 0, \frac{1}{2}) \\ \vec{r}_4 &= (0, \frac{1}{2}, \frac{1}{2}) \end{aligned} \tag{4}$$

For a simulation cell with n_c FCC cells in each direction, we can repeat the unit cell by the lattice vector $\vec{R}$

$$\vec{R} = (i, j, k) \text{ for } i = 0, 1, 2, ...n_c - 1, j = 0, 1, 2, ...n_c - 1, \text{ and } k = 0, 1, 2, ...n_c - 1 \tag{5}$$

to each of the position vectors. We then divide each of the four position vectors by n_c so that they are given a new fractional coordinates. For example, suppose we had a one-dimensional structure with 1 atom per cell at a position with $x = 0.5$. To create a simulation cell with 2 of these cells, we would create two positions at $x = 0.25$ and $x = 0.75$. As another example, if $n_c = 2$, we create a simulation cell with the atoms in an FCC structure but with a 4×8 unit cells $= 32$ atoms per unit cell.

We can construct a $n_c \times n_c \times n_c$ FCC unit cells in Python using arrays. We first introduce a vector r containing the $n = 4$ positions of the FCC atomic basis,

`r = [0,0,0; 0.5,0.5,0; 0.5,0,0.5, 0,0.5,0.5]`

We then add lattice vectors to the x, y, and z coordinates of the atoms in the cell. For a cubic material, the lattice vector has the form $\vec{R} = (k, l, m)a$, where a is the lattice length.

We loop over the indices of $\vec{R}$ and divide by n_c to ensure that the atomic positions are in the correct fractional coordinates in the simulation cell.

In the following code, we generate the (fractional) atomic coordinates of an FCC supercell (omitting the lattice parameter). The output of the following code is an array of fractional atomic coordinates of length $4 \times n_c \times n_c \times n_c$.

[5]:
```
## Code for generating an FCC supercell
import numpy as np

def fccmke(nc):
    """ Generate fractional coordinates of atoms in FCC supercell
       Input:
           nc (integer): supercell of nc x nc x nc dimensions
       Output:
           s (array): fractional atomic coordinates of supercell
    """

    natoms = 4
    r = np.array([[0, 0, 0], [0.5, 0.5, 0], [0, 0.5, 0.5], [0.5, 0, 0.5]])
    i1 = 0
    s = np.zeros((natoms * nc**3, 3))

    # in fractional coordinates
    for k in range(1, nc + 1):
        for l in range(1, nc + 1):
            for m in range(1, nc + 1):
                for i in range(natoms):
                    s[i1, 0] = (r[i, 0] + k - 1) / nc
                    s[i1, 1] = (r[i, 1] + l - 1) / nc
                    s[i1, 2] = (r[i, 2] + m - 1) / nc
                    i1 += 1
    return s

# Example usage:
nc = 3
s = fccmke(nc)
print(s)
```

```
[[0.         0.         0.        ]
 [0.16666667 0.16666667 0.        ]
 [0.         0.16666667 0.16666667]
 [0.16666667 0.         0.16666667]
 [0.         0.         0.33333333]
```

```
[0.16666667 0.16666667 0.33333333]
[0.         0.16666667 0.5       ]
[0.16666667 0.         0.5       ]
[0.         0.         0.66666667]
[0.16666667 0.16666667 0.66666667]
[0.         0.16666667 0.83333333]
[0.16666667 0.         0.83333333]
[0.         0.33333333 0.        ]
[0.16666667 0.5        0.        ]
[0.         0.5        0.16666667]
[0.16666667 0.33333333 0.16666667]
[0.         0.33333333 0.33333333]
[0.16666667 0.5        0.33333333]
[0.         0.5        0.5       ]
[0.16666667 0.33333333 0.5       ]
[0.         0.33333333 0.66666667]
[0.16666667 0.5        0.66666667]
[0.         0.5        0.83333333]
[0.16666667 0.33333333 0.83333333]
[0.         0.66666667 0.        ]
[0.16666667 0.83333333 0.        ]
[0.         0.83333333 0.16666667]
[0.16666667 0.66666667 0.16666667]
[0.         0.66666667 0.33333333]
[0.16666667 0.83333333 0.33333333]
[0.         0.83333333 0.5       ]
[0.16666667 0.66666667 0.5       ]
[0.         0.66666667 0.66666667]
[0.16666667 0.83333333 0.66666667]
[0.         0.83333333 0.83333333]
[0.16666667 0.66666667 0.83333333]
[0.33333333 0.         0.        ]
[0.5        0.16666667 0.        ]
[0.33333333 0.16666667 0.16666667]
[0.5        0.         0.16666667]
[0.33333333 0.         0.33333333]
[0.5        0.16666667 0.33333333]
[0.33333333 0.16666667 0.5       ]
[0.5        0.         0.5       ]
[0.33333333 0.         0.66666667]
[0.5        0.16666667 0.66666667]
[0.33333333 0.16666667 0.83333333]
[0.5        0.         0.83333333]
[0.33333333 0.33333333 0.        ]
[0.5        0.5        0.        ]
[0.33333333 0.5        0.16666667]
[0.5        0.33333333 0.16666667]
[0.33333333 0.33333333 0.33333333]
```

```
[0.5        0.5        0.33333333]
[0.33333333 0.5        0.5       ]
[0.5        0.33333333 0.5       ]
[0.33333333 0.33333333 0.66666667]
[0.5        0.5        0.66666667]
[0.33333333 0.5        0.83333333]
[0.5        0.33333333 0.83333333]
[0.33333333 0.66666667 0.        ]
[0.5        0.83333333 0.        ]
[0.33333333 0.83333333 0.16666667]
[0.5        0.66666667 0.16666667]
[0.33333333 0.66666667 0.33333333]
[0.5        0.83333333 0.33333333]
[0.33333333 0.83333333 0.5       ]
[0.5        0.66666667 0.5       ]
[0.33333333 0.66666667 0.66666667]
[0.5        0.83333333 0.66666667]
[0.33333333 0.83333333 0.83333333]
[0.5        0.66666667 0.83333333]
[0.66666667 0.         0.        ]
[0.83333333 0.16666667 0.        ]
[0.66666667 0.16666667 0.16666667]
[0.83333333 0.         0.16666667]
[0.66666667 0.         0.33333333]
[0.83333333 0.16666667 0.33333333]
[0.66666667 0.16666667 0.5       ]
[0.83333333 0.         0.5       ]
[0.66666667 0.         0.66666667]
[0.83333333 0.16666667 0.66666667]
[0.66666667 0.16666667 0.83333333]
[0.83333333 0.         0.83333333]
[0.66666667 0.33333333 0.        ]
[0.83333333 0.5        0.        ]
[0.66666667 0.5        0.16666667]
[0.83333333 0.33333333 0.16666667]
[0.66666667 0.33333333 0.33333333]
[0.83333333 0.5        0.33333333]
[0.66666667 0.5        0.5       ]
[0.83333333 0.33333333 0.5       ]
[0.66666667 0.33333333 0.66666667]
[0.83333333 0.5        0.66666667]
[0.66666667 0.5        0.83333333]
[0.83333333 0.33333333 0.83333333]
[0.66666667 0.66666667 0.        ]
[0.83333333 0.83333333 0.        ]
[0.66666667 0.83333333 0.16666667]
[0.83333333 0.66666667 0.16666667]
[0.66666667 0.66666667 0.33333333]
```

```
[0.83333333 0.83333333 0.33333333]
[0.66666667 0.83333333 0.5       ]
[0.83333333 0.66666667 0.5       ]
[0.66666667 0.66666667 0.66666667]
[0.83333333 0.83333333 0.66666667]
[0.66666667 0.83333333 0.83333333]
[0.83333333 0.66666667 0.83333333]]
```

1.3 A simple lattice sum

Let us consider the simple sum

$U = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\phi_{ij}(r_{ij})$

A simple model for the interatomic (pair-wise) potential is to assume the potential is a function of only the distance between the atoms. For now, let us assume the potential is the well-known Lennard-Jones potential of the form

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

where we introduce the parameters σ and ε that depend on the identity of the atom; σ is a the distance at which the potential is zero, i.e., $\phi(\sigma) = 0$ and ε is the absolute value of the minimum of the potential. For convenience, in the following code examples, we set $\sigma = \varepsilon = 1$.

Consider a simple lattice sum over N atoms in a cubic cell without periodic boundary conditions. We assume that we have a vector s containing the positions of the atoms in fractional coordinates (which must be multiplied by the cube side a). Let us sum over the Lennard-Jones interaction for an FCC supercell, as shown in the code below.

[ ]:
```
import numpy as np
import time # simple timer
```

[7]:
```
def lat_sum1(a, n, s):
    """ Naive implementation of lattice sum with Lennard-Jones potential

        Input:
          a (float): cell length
          n (integer): number of atoms in simulation cell
          s (array): fractional coordinates of atom positions
        Output:
          ucell (float): total energy of simulation cell
    """
    start = time.process_time()

    ucell = 0

    # sum over all atoms and divide by two
    for i in range(n):
        for j in range(n):
            xij = s[j, 0] - s[i, 0]
```

```
            yij = s[j, 1] - s[i, 1]
            zij = s[j, 2] - s[i, 2]
            dist = a * np.sqrt(xij**2 + yij**2 + zij**2)
            if dist > 0:
                phi = 4 * (1 / dist**12 - 1 / dist**6)
            else:
                phi = 0
            ucell = ucell + phi
    ucell = ucell / (2 * n)

    end = time.process_time() - start
    return ucell, end
```

This code is quite inefficient. `lat_sum.py` compute every pairwise interaction twice, and so must divide by 2. A simple improvement to our lattice sum code is to avoid overcounting with the equivalent expression.

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

An implementation of the above is shown in the code below.

[23]:
```
def lat_sum2(a, n, s):
    """ Avoid overcounting in lattice sums

        Input:
          a (float): cell length
          n (integer): number of atoms in simulation cell
          s (array): fractional coordinates of atom positions
        Output:
          ucell (float): total energy of simulation cell
    """
    start = time.process_time()

    ucell = 0
    # to avoid double counting, we change the indices over the loops
    # i will never equal j, so the if statement can be removed
    # no more factor of two!
    for i in range(n - 1):
        for j in range(i + 1, n):
            xij = s[j, 0] - s[i, 0]
            yij = s[j, 1] - s[i, 1]
            zij = s[j, 2] - s[i, 2]
            dist = a * np.sqrt(xij**2 + yij**2 + zij**2)
            phi = 4 * (1 / dist**12 - 1 / dist**6)
            ucell = ucell + phi
    ucell = ucell / n

    end = time.process_time() - start
```

```
    return ucell, end
```

1.3.1 Cutoffs

Often to speed up calculations, we choose to truncate the interatomic potential at some prescribed distance, referred to as the cutoff distance r_c.

We can implement a cutoff distance by including an `if` statement to test if the distances are less than the cutoff. This is implemented in the code below.

[27]:
```
def lat_sum3(a, n, rc, s):
    """ Inclusion of cutoff distance

        Input:
          a (float): cell length
          n (integer): number of atoms in simulation cell
          rc (float): cutoff distance
          s (array): fractional coordinates of atom positions
        Output:
          ucell (float): total energy of simulation cell
    """
    start = time.process_time()

    ucell = 0
    for i in range(n - 1):
        for j in range(i + 1, n):
            xij = s[j, 0] - s[i, 0]
            yij = s[j, 1] - s[i, 1]
            zij = s[j, 2] - s[i, 2]
            dist = a * np.sqrt(xij**2 + yij**2 + zij**2)
            if dist <= rc:
                phi = 4 * (1 / dist**12 - 1 / dist**6)
            else:
                phi = 0
            ucell = ucell + phi
    ucell = ucell / n

    end = time.process_time() - start
    return ucell, end
```

1.3.2 Periodic Boundary Conditions

To mimic an infinite system, we will use periodic boundary conditions in which the central simulation cell is replicated to fill space. In this instance, the lattice sum will be summing over the atoms in the cell and in nearby replica cells. Often, the cutoff distance is set such that we consider only cells adjacent to the central simulation cell.

As we did before in a naive implementation of periodic boundary conditions, we could sum over

neighboring cells explicitly.

[ ]:
```
# NOTE: this naive implementation is very slow
def lat_sum4(a, n, rc, c, s):
    """ Naive implementation of periodic boundary conditions

        Input:
          a (float): cell length
          n (integer): number of atoms in simulation cell
          rc (float): cutoff distance
          c (integer): number of periodic neighbors to left and right
          s (array): fractional coordinates of atom positions
        Output:
          ucell (float): total energy of simulation cell
    """
    start = time.process_time()
    ucell = 0
    for i in range(n):
        for j in range(n):
            for k in range(-c, c + 1):
                for l in range(-c, c + 1):
                    for m in range(-c, c + 1):
                        xij = k + s[j, 0] - s[i, 0]
                        yij = l + s[j, 1] - s[i, 1]
                        zij = m + s[j, 2] - s[i, 2]
                        dist = a * np.sqrt(xij**2 + yij**2 + zij**2)
                        if 0 < dist <= rc:
                            phi = 4 * (1 / dist**12 - 1 / dist**6)
                        else:
                            phi = 0
                        ucell = ucell + phi
    ucell = ucell / (2 * n)

    end = time.process_time() - start
    return ucell, end
```

Again, we can improve this code by noting that atoms on one side of a cell are unlikely to interact (significantly) with atoms in a neighboring cell on the opposite side. Thus, to avoid excess computational cost, we want to avoid calculating distances to atoms that are beyond the cutoff distance.

One strategy to accomplish this is to consider an imaginary box centered around the atom for which one is summing the interactions. If the box is set up so that its side has a length that is twice the cutoff distance (i.e., on both sides), than all the atoms that can interact with the atom at the center are within the box. One then only sums over the atoms within the box. One still needs to check the cutoff criteria for cases such as atoms at the box corners which will be outside of the cutoff distance. More sophisticated methods, such as neighbor lists as discussed in the text, may be used

for large systems.

For the small systems considered here, we employ a method called the *minimum image convention*. We take the size of the imaginary box to be the size of the simulation, so that the cutoff distance r_c must be less than or equal to half the size of the simulation cell. This way, each particle interacts only with the nearest image of the other particles of the simulation cell. Using the minimum image convention reduces the number of particles to search. Cutoff distances less than half the box size can be used.

One reason the minimum image convention is an attractive strategy is its ease of implementation. Assume we have a simulation cell with side of length a and we want to find the neighbors of atom i. We first pick another atom j and compute r_{ij}. We check to see if each component of the r_{ij} lies inside or outside the cell with side a centered on atom i. If the component is outside that cell, we add or substract a lattice vector to bring it back within the cell, which will be accomplished with the `round(x)` function. We create the image using the function

```
x = x - round(x)
```

Let us examine what `x = x - round(x)` does. If $x > 1/2$, x is replaced by $x - 1$. If $x < -1/2$, x is replaced with $x + 1$. Otherwise, x is unchanged. Thus, this function selects only the nearest image of atom i.

We can implement the minimum image convention by replacing the sum over the lattice vectors.

[31]:
```
def lat_sum5(a, n, rc, s):
    """ Implementation of minimum image convention

        Input:
          a (float): cell length
          n (integer): number of atoms in simulation cell
          rc (float): cutoff distance
          s (array): fractional coordinates of atom positions
        Output:
          ucell (float): total energy of simulation cell
    """
    start = time.process_time()
    print("starting lattice sum")

    ucell = 0
    for i in range(n - 1):
        for j in range(i + 1, n):
            xij = s[j, 0] - s[i, 0]
            yij = s[j, 1] - s[i, 1]
            zij = s[j, 2] - s[i, 2]
            xij = xij - round(xij)
            yij = yij - round(yij)
            zij = zij - round(zij)
            dist = a * np.sqrt(xij**2 + yij**2 + zij**2)
            if 0 < dist <= rc:
                phi = 4 * (1 / dist**12 - 1 / dist**6)
```

```
            else:
                phi = 0
            ucell = ucell + phi
    ucell = ucell / n

    end = time.process_time() - start
    return ucell, end
```

1.4 Lattice statistics

The lattice sums implemented above are the basis of a set of techniques call *lattice statistics*. In these simulations, the temperature is 0 K and the equilibrium atomic structure is determined by finding the positions that minimize the potential energy of the system. For example, we could take all the atoms in the positions of a perfect solid and vary the lattice parameters to find the equilibrium structure. We can also take a perfect solid and create a vacancy by removing one atom, and subsequently find the structure and energy of that vacancy. We can also vary the volume and plot the energy per unit volume in which that slope of the resulting curve is the negative of the pressure at 0 K.

Such calculations typically involve lattice sums to evaluate the energy and minimization routine to vary the atomic positions and/or lattice parameters until the minimum energy configuration is reached. We shall see a few of these cases in the exercises below.

2 Exercises

2.1 Perfect cubic solids

1. Create a code to calculate the energy of a solid with interactions described by the Lennard-Jones potential. If appropriate, assume the minimum image convention. The input parameters should be the lattice constant a_0, the number of repeated units n, and the cutoff distance r_c. The resulting lattice constant is $a = na_0$. Assume an FCC lattice and that $\$\varepsilon = \sigma = 1\ \$$ in the Lennard-Jones potential.
2. Vary the lattice constant and find the corresponding minimum energy and equilibrium lattice parameter.
3. Vary the cutoff distance and find the associated error in the total energy calculations and lattice parameter for at least two values of the cutoff distance. Be sure to vary the cutoff such that you include different numbers of atomic shells in the sums. Create a plot of these values as a function of the number of neighboring shells. Discuss the differences you find with respect to the results with infinite sums.
4. Repeat problems #1-3 for the body-centered (BCC) structure. Note that the lattice positions will be different than those in FCC case. Determine the equilibrium energy for the BCC system and compare with that of the FCC system. Which structure is more stable (at 0 K)?
5. For the FCC lattice, vary the lattice parameter and plot the energy vs volume. From this plot, determine the pressure and Gibbs free energy at 0 K. Plot the volume and Gibbs free energy as a function of the pressure.