PHAS0030: Computational Physics
Session 7: Modelling with Particles

David Bowler

In this session, we will explore how particle models of matter in many different forms and on different length scales can be used to understand various properties of systems. The important class of integrators known as Verlet methods will be introduced and used to explore the properties of the simple Lennard-Jones model of inter-particle interactions. We will also consider longer-range interactions such as gravity and electrostatics, and touch on the general idea of coarse graining.

1 Objectives

The objectives this session are to:

- Introduce the Verlet integrator and discuss energy conservation
- Discuss simulation cells and boundary conditions
- Use the Lennard-Jones potential as an example of particle-particle interactions
- Examine appropriate initial conditions for simulations
- Consider what quantities can be averaged and calculated
- Discuss inverse square law potentials (gravity and electrostatics)
- Briefly introduce general ideas of coarse-graining

2 Review of Session 6

In the sixth session, we tackled the classical wave equation, finding that a second-order centred finite difference approach was accurate enough, and noting that the wave speed, grid spacing and time step are all closely related. We saw that solving in two dimensions was an almost trivial extension of one dimension. We then turned to the quantum mechanical Schrödinger equation, with the time-dependent equation requiring an implicit solver for stability. We also introduced complex numbers in Python to solve this equation. We then saw that the time-independent equation could be solved with the approaches we used for ordinary differential equations, and used the shooting method to find eigenvalues that allowed our solutions to match the necessary boundary conditions.

3 Interactions between particles

If we are interested in calculating the properties of materials, we can make significant progress by modelling the particles[1] that make up the material as classical particles, with their time evolution determined by Newton's laws of motion.

We can use this approach for many different problems, including: solids, liquids and gases, where 'particle' will mean the atoms or molecules that make up the material; solutions and plasmas, where 'particle' means positive and negative ions or ions and electrons; granular materials, like sand; and even astrophysical systems where each 'particle' is a planet or a star. We can also 'coarse grain' a system, so that we model appropriate units (e.g. ferromagnetic domains, or elements of proteins) without the full details of the atoms or molecules that make them up.

The interactions will depend on the system: for atoms and molecules, we will use a classical inter-particle potential which approximates the detailed quantum mechanical interactions; for charged particles and astrophysical systems, we use inverse square interactions (electrostatic or gravity as appropriate); in other situations, we can parameterise an interaction based on the important physics.

In every case, we will need to calculate the energy of the system, and forces on the particles; the functional forms of these quantities will depend on the detailed physics, and will be part of the setup for the problem. In most cases, we will then allow the system to develop in time (using a relatively simple differential equation which we integrate numerically) and calculate average or time evolving properties. The properties of interest will depend on the system, and we will discuss them below. When modelling materials, it is often important to calculate total energies for stable or metastable structures; in these cases, the calculation will involve moving the atoms or molecules until the forces are zero.

3.1 Calculating energy and forces

In many situations, the energy and forces can be written in terms of *pairs* of particles[2], in particular as a function of the distance between particles, $\mathbf{r_{ij}}$. This is what we will use in all the exercises for this session; to calculate the energy, you will need to have loops like this:

```python
Etot = 0
for i in range(Npart):
    for j in range(i+1,Npart):
        dr = r[j] - r[i] # NB dr is a length 3 array
        Etot = Etot + calc_energy(dr)
```

[1] This is used as a general term which we will define soon.

[2] This is not always true, and in some cases the angles (three-body) and twist (four-body, or dihedral) are used.

We exclude the case when i==j in this case because the particle does not interact with itself[3]. Notice how we have made the loop over j start from i+1: this automatically excludes self-interaction, and avoids counting the energy for each interaction twice.

For the forces, you should note that, with a pair-wise interaction, the force on particle i from particle j is just the negative of the force on particle j from particle i. This can help to make calculations more efficient. For the calculation of forces, we can write code like this:

```
forces = np.zeros((Npart,3)
for i in range(Npart):
    for j in range(i+1,Npart):
        dr = r[j] - r[i]
        fij = calc_force(dr)
        forces[i] += fij
        forces[j] -= fij
```

Note that the force is often defined in terms of $\mathbf{r}_{ij} = r_j - r_i$ which explains the order of the indices in the call to the routine calc_force.

3.2 Indexing

It is worth thinking a little carefully about indexing, as this is a complex situation. We will often want to store the positions and velocities for all particles at every timestep, and each of these quantities is a vector (in 2D or 3D, depending on the problem). The indexing, and a loop over time, should look something like this:

```
Nsteps = 500 # Number of timesteps
Npart = 64   # Number of particles
pos = np.zeros([Nsteps, Npart, 3])
vel = np.zeros([Nsteps, Npart, 3])
mass = ...
# Set initial positions and velocities
pos[0] = ...
vel[0] = ...
for timestep in range(0,Nsteps-1):
    E = calc_energy(pos[timestep])
    acc = calc_forces(pos[timestep])/mass
    pos[timestep+1] = # Update based on force and velocity
```

Note that pos[timestep] is a NumPy array of size $N_{part} \times 3$. Of course, you can calculate both energy and forces in the same routine, and return them using something like return Etot, forces, which will be more efficient.

[3] Though there are some situations where there is an energy associated with a particle itself, in which case we can add it *before* the loop over j

4 Small particles: atoms and molecules

There are three key ideas that we will have to introduce when modelling systems of atoms or molecules: the integration scheme that we use (and why it is important); the simulation cell (a box within which we run the calculation); and the interaction between particles (we will use a standard example, the Lennard-Jones potential).

Given an potential for the interaction between particles (which we will come to below), we can calculate the total energy of the system, and the force on each particle. If we then integrate the Newtonian equations of motion over time, we have the Molecular Dynamics (MD) method. In this case, we are using the time evolution to sample different arrangements of the particles (different states of the system), over which we can then perform an average. This corresponds rather closely to many experiments, where a system is set up, and then a series of measurements are made over time to find an average value[4].

We can calculate thermodynamic properties of the system from these averages, using the ideas of statistical mechanics. Typically we will either need to run for a long time, to generate uncorrelated arrangements of the atoms, or do many short runs starting from different (random) initial conditions. Each run is called a trajectory. When performing simulations, we conserve certain quantities: almost always the number of particles; frequently the volume of the system (though sometimes we conserve a pressure and allow the volume to vary); and either total energy or temperature. Conserving particle number, volume and energy (NVE) is known as the microcanonical ensemble (roughly speaking, there is an equal probability for each energy state); conserving particle number, volume and temperature (NVT) is known as the canonical ensemble (a system in thermal equilibrium with a heat bath).

If we are interested in the structure of a bulk crystal, a surface or a molecule, then we can search for minimum energy (often using some of the approaches we discussed in Session 2, such as conjugate gradients). This is known as structural relaxation.

4.1 The Verlet integration scheme

The schemes that we have used so far to integrate equations of motion (i.e. differential equations) are accurate, but do not necessarily conserve energy, nor indeed are they time reversible. These are two key properties that we require to perform accurate molecular dynamics, and are obeyed well by a set of integration schemes known as Verlet integrators.

(It is worth mentioning at this point another concept: we consider a space defined by a particle's position and momentum, known as phase space. For a particle in one dimension, this can be represented by a plot of the velocity against its position; the kinetic energy will

[4] There is another approach to sampling which we will consider in Session 8: to choose different configurations of particles at random and average over them based on some measure of how likely they are to be found. When both approaches are properly converged, they should give the same results.

be determined by the velocity, and the potential energy by the position, so an integrator that conserves energy will stay on a surface of constant energy in phase space. These integrators are known as *symplectic* and preserve the area of phase space.)

The simplest Verlet scheme is based on a Taylor expansion, and calculates the position and velocity out of order: the velocity update relies on the position update.

$$\begin{aligned} x(t+\Delta t) &= 2x(t) - x(t-\Delta t) + \Delta t^2 \frac{F(x(t))}{m} \\ v(t) &= \frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t} \end{aligned}$$

where $F(x(t))$ is the force calculated based on the position $x(t)$. These equations are trivially generalised to vector equations.

The form that we will use[5] is the velocity-Verlet integrator, which allows for the evaluation of position and velocity at the same step:

$$\begin{aligned} x(t+\Delta t) &= x(t) + \Delta t\, v(t) + \Delta t^2 \frac{F(x(t))}{2m} \\ v(t+\Delta t) &= v(t) + \Delta t \frac{F(x(t)) + F(x(t)+\Delta t)}{2m} \end{aligned}$$

In this case, the velocity update requires that the force based on the new positions is calculated first (while the old force is kept). For all of these equations, the time reversibility should be clear: change the sign of Δt, and the equations reverse exactly.

4.2 Simulation cells

Most if not all experiments which involve thermodynamic quantities involve a finite sample, which might be enclosed by some volume. We will do the same thing in our computer simulations: we define a box (often called the *simulation cell*[6]) into which we will place some particles and allow them to evolve in time.

But it is not enough simply to define a box, as you will find in Question 4 of the exercises: we need some kind of boundary condition. The two most common conditions are hard

[5] The final common form is the leap-frog Verlet method, which is very similar to both of the schemes given here, but calculates the velocity at half-timesteps, $v(t + \Delta t/2)$.

[6] People often refer to this as a unit cell: I think that this is misleading, as we use this term in crystallography for the repeat unit of crystalline solids, where the term simulation cell is unambiguous.

walls (which you will implement in the further work) and periodic boundaries (which you will investigate in Question 5)[7].

Hard walls impose elastic collisions with the walls, so that if a position update takes the particle outside the box, it is reflected back into the box with reversed velocity. This is a useful approach if studying gases in confined containers, for instance. More common are periodic boundaries, where a particle that goes outside the box is simply wrapped back around to the other side of the box (easily implemented computationally using modulo) without changing the velocity.

4.3 Exercises

Note that *all* these exercises involve just *one* particle; we'll move on to multiple particles after the next section. You could have arrays like `pos = np.zeros((Nsteps,1,3))` if you want, or just have `pos = np.zeros((Nsteps,3))`.

4.3.1 In-class

1. Single particle, no box

   a. Create arrays to store the position, velocity and acceleration of a particle in three dimensions for `Nstep` steps (set `Nstep` to 200; your arrays should be `Nstep` $\times$ 3 in size). Initialise: $\mathbf{r} = (1{,}0{,}0)$, $\mathbf{v} = (0{,}1{,}0.1)$ and $\mathbf{a}(t) = (-1{,}0{,}0)$. Set the timestep `dt` to 0.1

   b. Now write a simple `for` loop to implement the velocity Verlet algorithm given above, running from step 1 and covering `Nstep` steps. At each step, calculate the time, and use $\mathbf{a} = (-\cos(t), -\sin(t), 0)$.

   c. Using `fig = plt.figure` and `ax = fig.add_subplot(111,projection='3d')`, plot the trajectory of the particle in 3 dimensions with `ax.plot(x,y,z)` where `x, y, z` are given by components of $\mathbf{r}$ (you might use `r[:,0]` etc).

2. Single particle, open boundaries.

   a. Define a side length, L, for a box (say 10 units), and set up position and velocity arrays as before (we will assume that the acceleration is zero). Choose a random starting point[8] for the particle in the box (use `np.random.rand(3)` which will return a length 3 array with numbers drawn at random from a uniform distribution between 0 and 1 and scale by box length) and random initial velocity (this time do not scale `np.random.rand`).

[7] The other type of boundary condition that is encountered is an open boundary, though this is incompatible with particles that will propagate outwards without any restriction.

[8] It is important to note that computer-generated "random" numbers are actually *pseudo-random*; we will discuss this in much more detail in Session 8.

   b. Use a simple `for` loop to evolve forward in time for 200 steps using velocity Verlet again, and plot the path of the particle using `ax.scatter` in three dimensions.

   c. Where is the particle relative to the edges of the box at the end of the simulation?

3. Single particle, periodic boundaries.

   a. Set up a box as in the previous part, along with arrays for position and velocity, and choose a random starting point and velocity.

   b. Write another `for` loop as in Q2 above but this time implement periodic boundary conditions: for each component of position, you need to ensure *after updating the position* that if `r[i]`<0 or `r[i]`$> L$ then that component is wrapped back into the box (you could write a function to do it, or just apply the modulo operator % giving `r%boxlen` or something similar).

   c. Plot the trajectory again, and check that the kinetic energy is conserved (you can use something like `np.sum(v*v,axis=1` on the velocity array, and plot it against time).

4.3.2 Further work

1. For the single particle in Q1 in-class, experiment with other plots: create two 3D subplots, and plot the particle trajectory in the first with `ax.scatter(x,y,z)`. In the second, use array slices (`x[::n]` with extract every nth point in an array) and plot the velocity vectors using `ax.quiver(x,y,z,vx,vy,vz)`. You might want to plot the trajectory using `ax.plot(x,y,z` as well.

2. Calculate the kinetic energy (you can use something like `np.sum(v*v,axis=1` if you have used the array structure suggested above—this will give an array of the kinetic energy at each step) and plot it. How well is it conserved? If you have time, experiment with the effect of timestep on conservation.

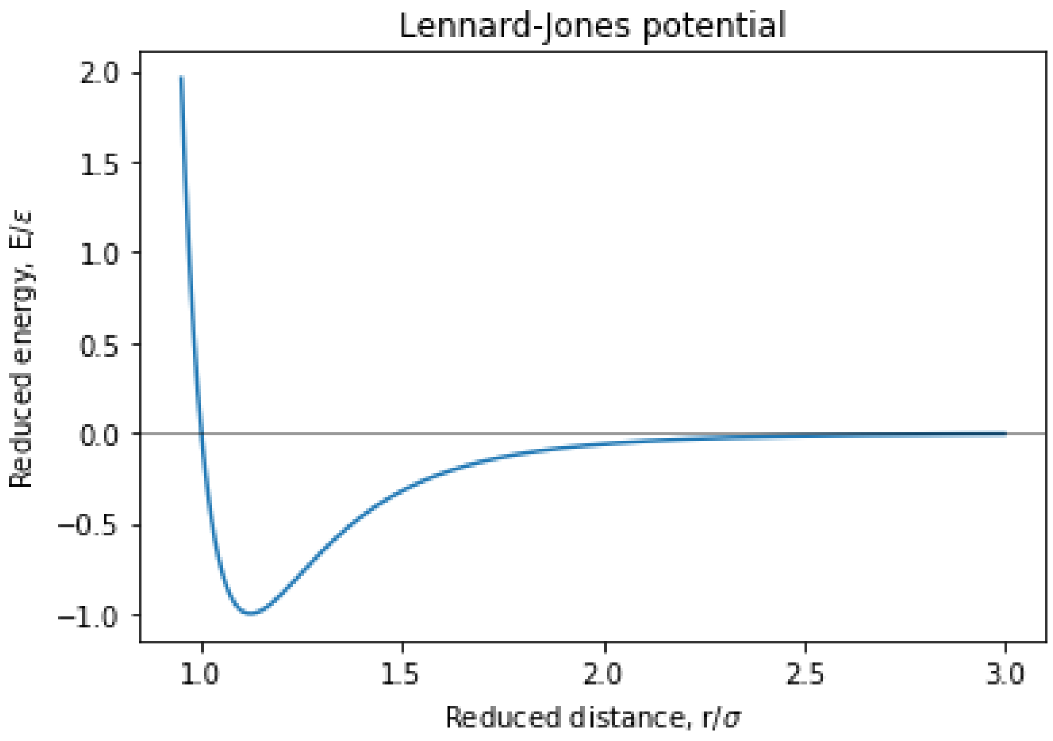
3. Write a function to apply hard-wall boundary conditions: $r_i > L \Rightarrow r_i = 2L - r_i, v_i = -v_i$ and $r_i < 0 \Rightarrow r_i = -r_i, v_i = -v_i$. Run the same calculation that you did for Q4 of the in-class exercises, but apply your hard-wall function after updating the position. Plot the path of the particle in three dimensions, and check that the conditions are applied correctly. (If you use `%matplotlib notebook` then the 3D plot will be interactive, and you can change the viewpoint with your mouse.) You could also check that the kinetic energy is conserved again.

5 The Lennard-Jones potential

The Lennard-Jones potential is commonly used to describe interactions between neutral atoms and molecules. It is often written as:

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

where the first, repulsive, term approximates the Pauli repulsion felt at short distances when the electron clouds of the particles overlap while the second, attractive, term approximates the van der Waals interaction, which is long-ranged. The potential is shown below, in terms of the constants σ and ϵ.



The Lennard-Jones potential

The potential crosses zero when $r = \sigma$ and has a minimum at $r = 2^{1/6}\sigma$. The interaction between neon atoms can be described with a Lennard-Jones potential with $\sigma = 0.275$nm and $\epsilon = 36k_B$ (so that the binding energy is equivalent to a temperature of 36K). The mass of a neon atom is $20.2m_P$ with $m_P = 1.673 \times 10^{-27}$kg. To calculate the *potential* energy of the system, you would use the kind of loop over pairs of particles we wrote above:

```
Epot = 0
for i in range(Npart):
    for j in range(i+1,Npart):
        dr = r[j] - r[i] # NB dr is a length 3 array
        Etot = Etot + calc_LJ_energy(dr)
```

The kinetic energy of the system can be found from the particle velocities, and the total energy at any given timestep is the sum of the two.

Notice that we have to be careful when using periodic boundaries: each particle is effectively repeated multiple times, at different positions. For this course, we will simply ensure that the particle-particle interaction is calculated for the *smallest* distance (modulo box length). This is easily done like this: `dr -= boxlen*np.rint(dr/boxlen)` where `np.rint` rounds to the nearest integer. There are more subtle points to be understood with respect to cutoffs and boundary conditions, but this will suffice for now.

We can of course calculate the force on a particle from another particle by differentiating the potential *analytically*:

$$\begin{aligned}\mathbf{F}_i(\mathbf{r}_{ij}) &= -\nabla_j V(r_{ij}) = -24\epsilon\,\hat{\mathbf{r}}_{\mathbf{ij}}\left[2\frac{\sigma^{12}}{r_{ij}^{13}} - \frac{\sigma^6}{r_{ij}^7}\right] \\ &= 24\epsilon\mathbf{r}_{ij}\left[\frac{\sigma^6}{r_{ij}^8} - 2\frac{\sigma^{12}}{r_{ij}^{14}}\right] = -\mathbf{F}_j(\mathbf{r}_{ij})\end{aligned}$$

Note that this is the force on particle i from particle j (and minus the force on j from i as shown); the total force on particle i will involve summing up the forces from all particles in the simulation.

Initialisation is a key part of a simulation, and requires some care. We will use a cubic crystal structure to ensure that the atoms are placed at reasonable distances from each other, and then draw the initial (random) velocities from a uniform distribution so that there is some initial temperature to ensure dynamics.

As with any experiment, once a simulation has been set up, some time is required for it to equilibrate (come into equilibrium): to reach a state which is uncorrelated from the starting state. Once a system is in equilibrium, we can sample appropriate data and average to find properties of the system. For fluids, the radial distribution function (a measure of the average density of particles at a given distance from another particle) can be easily calculated, and directly related to neutron scattering data, for instance.

Once we have run a calculation for an appropriate amount of time, we can measure, for instance, the distribution of velocities. Thermal physics (or statistical mechanics) can be used to show that the velocity distribution of particles at temperature T should be:

$$\begin{aligned}p(v) &= Av^2 e^{-mv^2/2k_BT} \\ A &= \sqrt{\frac{2}{\pi}\left(\frac{m}{k_BT}\right)^{3/2}}\end{aligned}$$

But how do we know the temperature? The average kinetic energy *per particle*[9] can be shown to be:

$$\frac{1}{2}mv^2 = \frac{3}{2}k_BT$$

[9] We should actually consider the average per degree of freedom; for a system with zero net momentum, this is $3N_{Part} - 3$, but the difference is rather small.

so we can calculate the temperature of the system from the average kinetic energy.

Realistic simulations require much larger systems than we have used here, and considerable care to ensure equilibration. Typically simulations will range from thousands to millions of atoms, and times will range from picoseconds to microseconds. The timestep that is used will be of the order of 1fs ($=10^{-15}$s) so long time simulations are extremely expensive in terms of computer time.

5.1 Exercises

5.1.1 In-class

1. Write a function to calculate the Lennard-Jones potential energy for a set of N_{Part} particles; you should pass the positions (in three dimensions), the number of particles, N_{Part}, and the constants `sigma` and `epsilon` as parameters. You should assume that the position array is of dimension ($N_{Part} \times 3$). You will need a *double* `for` loop, with the first index `i` running from 0 to $N_{Part} - 1$ and the second index `j` running from `i+1` to N_{Part}. Calculate $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ and then evaluate r_{ij}^2 with `np.sum(rij*rij)`.

2. Write a function to evaluate the Lennard-Jones force on a set of N_{Part} particles, following the same approach as in Q1. You can also calculate the energy in this loop if you want. (Note that the loops will be the same, but now when you calculate a force between particles `i` and `j` this gives *both* $F_i(r_{ij})$ *and* $-F_j(r_{ij})$.)

3. Initialise variables to represent ϵ, σ, k_B, m. Set a number of steps, N_{step} to 500 initially.

4. Now we are going to create cubic lattice by repeating the basic building block in all three directions. The building block is a cube with one particle in the bottom left corner. Set a variable N_{cell} to represent the number of cubes along each axis (I suggest 2 or 3 to start) and set $N_{Part} = N_{cell}^3$. Create arrays to hold position and velocity for N_{Part} particles and N_{step} timesteps (they should have dimension $N_{step} \times N_{part} \times 3$ for ease of reference). Set the box length to $L = 2^{1/6}\sigma N_{cell}$. Loop over N_{cell} in x, y and z with variables `i`, `j` and `k`, and set the position of each atom to $L \times (i, j, k)/N_{cell}$, for instance:

```
part_no = 0
for i in range(Ncell):
    for j in range(Ncell):
        for k in range(Ncell):
            pos[part_no] = (boxlen/Ncell) * np.array([i, j, k])
            part_no += 1
```

   Plot your atoms in 3D to check.

5. We will initialise the velocities: use `np.random.normal(size=(Npart, 3))` to generate a normal distribution, and then rescale them so that the temperature is correct (you will have to rearrange $mC^2 \sum_i v_i^2 = 3N_{part}k_BT$ to find the rescaling factor C; use `np.sum(v*v)` to evaluate the sum in this case). Set up arrays to store the potential and

kinetic energies, and the temperature, for all steps. Calculate the potential energy and forces for the initial set of atomic positions using your functions from Q1 and Q2. Calculate the accelerations for this time step (`acc_this`) from the forces. Store the potential energy for step zero, and calculate the total kinetic energy and temperature for the initial velocities and store these.

6. Now use the velocity Verlet algorithm to propagate your system. At each step you will need to evaluate `acc_next` after you calculate the updated positions, and use `acc_this` and `acc_next` to calculate the new velocities. Store the potential energy, kinetic energy and temperature at each step. Plot the two energies and their sum against time: the total should be conserved. (You may find it useful to normalise the energies by dividing by $N_{part}\epsilon$.)

5.1.2 Further work

1. Plot snapshots of the atomic configurations at different times to get a sense of how the atoms move.
2. Plot the total trajectory of one or two atoms.
3. Plot a histogram of the particle speeds for the final step (use `np.sqrt(np.sum(v*v,axis=1))` or something similar) and, on the same graph, plot the expected distribution given the temperature at the final step.

6 Inverse quadratic interactions

Both gravitational and electrostatic interactions depend on the inverse square of the distance separating the particles. This interaction is long-range, and requires considerable care as the number of particles increases. This type of problem occurs when modelling plasmas (e.g. in nuclear fusion experiments) and galaxies (where stars are the particles, but there can be a continuous distribution of dust that must be included). Summing over every pair of interactions explicitly is expensive, and there are better methods (we will discuss fast Fourier transforms in Session 9).

We will apply our velocity Verlet scheme to a simple gravitational problem to illustrate how this might be used. Trajectories are extremely important in many areas of astrophysics and space science, and it may be that a more accurate integrator is needed than Verlet (sacrificing energy conservation for position accuracy may be appropriate). In these cases, we are interested in the details of the trajectory, rather than the averages we looked at for molecular dynamics.

In our example, we will consider the Earth-Moon system (though with proper physics, rather than the crude approximation that we used in Session 2) and will add a geostationary satellite. We can define the following constants (note that we will only consider values to one decimal place):

$$\begin{aligned}
G &= 6.7\times 10^{-11} m^3 kg^{-1} s^{-2} \\
M_E &= 6.0\times 10^{24} kg \\
M_M &= 7.4\times 10^{22} kg \\
M_S &= 1\times 10^{3} kg \\
R_{EM} &= 3.8\times 10^{8} m \\
R_{ES} &= 4.2\times 10^{7} m
\end{aligned}$$

The algorithm required is essentially the same as we used for the neon atoms, in Section 5, though we now have to account for varying masses. This simple kind of approach can be applied to significantly larger systems; a recent paper (Phys. Rev. E **99**, 022125 (2019) available here) models the evolution of structures such as spiral arms in galaxies, considering 100,000+ particles (though older simulations, such as Math. Modell. **9**, 785 (1987) available here, showed that basic forms can be seen even with a few hundred stars).

6.1 Exercises

6.1.1 In-class

1. Write a function to evaluate the *acceleration* due to the gravitational force between N_{body} bodies, passing arrays of position and mass along with N_{body} as parameters. You will need to loop over all pairs, excluding `i==j`. Remember that the force on `i` from particle `j` is $F_i = Gm_im_j\mathbf{r}_{ij}/r_{ij}^3$, so the acceleration will just remove the factor of m_i.

2. Now, for `Nbody=3`, set up arrays for mass, position and velocity (the last two should be of dimension $N_{step}\times N_{body}\times 3$ for $N_{step}=300$). Set the initial positions with the Earth at the origin and the moon and the satellite arranged along the x axis. The initial velocities for the moon and the satellite should be along the y axis (with values $v = r\omega$, calculating ω from the periods of the moon and satellite of 24 hours and 28 days, respectively). Set a timestep of one hour (initially). Calculate the initial accelerations using your force function from Q1.

3. Now propagate the system using a velocity Verlet approach, and plot the trajectories in 3D using `plt.figure`. You may want to plot the Moon and satellite trajectories relative to the Earth's position for clarity.

6.1.2 Further work

1. Start the satellite with a velocity along the z axis instead of the y axis, and compare the two trajectories. Does the change of plane (and hence change of distance to the moon) make any difference?

7 Coarse-graining

The approach we have seen so far of modelling a system based on its constituent parts is extremely powerful. In particular, it enables us to reach larger scale systems and longer time scales by concentrating on one set of interactions (e.g. we ignore the electrons when considering neon atoms). If a system can be approximated with “lumps” of matter (on whatever scale) then given an appropriate interaction we can examine its properties. The interaction will generally need both an attractive and a repulsive part, which will naturally result in an equilibrium spacing where the energy is at a minimum.

We will consider a final example of a chain, fixed at both ends, under the influence of gravity (this is a 2D problem). The links of the chain will be approximated by particles, and the interaction between links with a form like this:

$$V(r) = D\left(\frac{1}{r^4} - \frac{1}{r^2}\right)$$

We will start with the chain horizontal (i.e. $y = 0$ for all values of x, where x is distance along the chain). Under the influence of gravity it will start to fall, until the links are stretched too far, at which point it will rebound. To find an equilibrium structure we need to remove energy from the system somehow: the effects of friction etc can be easily modelled simply by reducing the velocity at each step ($v = 0.99 \times v$). The force due to the interaction is given by:

$$\mathbf{F}(\mathbf{r}) = \mathbf{r}D\left(\frac{2}{r^4} - \frac{4}{r^6}\right)$$

7.1 Exercises

7.1.1 In-class

1. Write a function to calculate the interaction *force* above. You should take $D = 10^4$.

2. For a chain with 20 links create arrays to store the position and velocity for N_{step} time steps (set this to 3000). Set a timestep of $1e-3$s, and space the links in the chain (the particles) $\sqrt{2}$m apart, and give the links a mass of 1kg each. Set the velocities to zero, and calculate the initial acceleration from your interaction force function above and the acceleration due to gravity (9.8ms^{-2}). Ensure that the acceleration and velocity for the first and last links are zero (these are fixed).

3. Propagate your chain in time using the velocity Verlet algorithm, and plot its displacements every 200 or 250 steps. Does the behaviour seem reasonable?

7.1.2 Further work

1. Introduce a damping factor into your integration scheme, testing the effect of different values between 0.9 and 1.0.

8 Assignment

You will be performing a molecular dynamics simulation of the Lennard-Jones model of Ne, using the parameters given above in Section 5.

1. Write a function to calculate the Lennard-Jones energy and force (or, if you prefer, write two functions). You may take this from the in-class exercises, but make sure there is a *clear* docstring and good comments.

2. Based on the in-class exercises, set up a lattice of 64 Ne atoms on cubic grid points, using a box length of $1.1 \times 2^{1/6}\sigma$ (slightly larger than we used in class). Plot your system in 3D with a scatter plot.

3. Initialise the velocities from with a normal distribution using `np.random.normal`, and rescale so that they give the system an initial temperature of 50K, using the fact that the average kinetic energy per particle is $3k_BT/2$.

4. Create an array to store the mean squared displacement (a single number) at each step of the simulation. Now run the simulation for 10,000 steps with a timestep of 10^{-15}s (this make take several minutes; you might want to run a shorter test first to ensure that your code is functioning correctly), storing at least the mean square displacement and velocity at each step. Ensure that you apply periodic boundary conditions (as we did in class). (The mean squared displacement is defined as:

$$D(t) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{r}_i(t) - \mathbf{r}_i(0))^2$$

5. which can be calculated efficiently with `np.sum` and simple numpy array arithmetic; note that it will have 9,999 non-zero elements.) Plot the displacement against time. Now plot $D(t)/(6t)$ against time from 2ps onwards (i.e. use array slices like `[2000:]`). Does it converge to a constant value?

6. Now calculate the following integral:

$$I(t) = \frac{1}{3N}\int_0^t \sum_{i=1}^{N} \mathbf{v}_i\,(t) \cdot \mathbf{v}_i(0) dt$$

7. with a simple numerical rectangle integration. Plot this quantity, which should converge with time. Estimate the converged value, and see how well it compares to $\frac{1}{6}$ of the value from the previous question: when fully converged, the two should agree. (The integral $I(t)$ is known as an auto-correlation function and is used widely in MD analysis.)

8. Write a brief conclusion, touching on the convergence of these quantities and the effort required to calculate both.

9. You could explore longer times, or the effect of the size of the system, if you wanted to extend the simulation.

9 Progress Review

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- Write a velocity Verlet integrator
- Initialise a set of particles to a cubic lattice and appropriate temperature
- Propagate the particles and monitor energy and temperature
- Use the Verlet integrator to propagate small numbers of particles with inverse square interactions
- Understand the principle that coarse-graining allows modelling of systems at appropriate length scales.