ASSIGNMENT 15: SOLUTIONS

E. ARNOLD, M-S. LIU, R. PRABHU & C.S. RICHARDS
THE UNIVERSITY OF CAMBRIDGE

Written in XeLaTeX

INTERACTING MONTE CARLO

INTRODUCTION

In previous tutorials, we have simulated the trajectories of an adsorbate diffusing by jumping on the substrate surface layer. This tutorial explores what happens when there are more than one single adsorbate and the adsorbates interact with each others. However, writing an interacting Monte Carlo (MC) simulation from scratch is a difficult task and is beyond the scope of this tutorial. Instead, we will provide the reader to an already-written simulation, and instruct the reader to analyse jump diffusing systems.

We will first explain the theory behind the simulation, and look at how the radial distribution function (RDF) is different for adsorbates on a 2D surface. The tasks will ask the reader to produce graphs using the simulation and finally write a script that calculates the RDF.

KINETIC MONTE CARLO

When simulating a system of particles, the most straight forward method would be to analytically calculate the trajectories of the particles at each timestep. However, this is not computationally feasible for a system with a huge number of adsorbates. Moreover, using finite timesteps cannot resolve interactions of short time lengths ($10^{-15}s$) such as atomic vibration [1] over longer simulations. Therefore, we need to advise a more efficient simulating algorithm.

The kinetic Monte Carlo method (KMC) is used in this tutorial to simulate the evolution of a system of adsorbates jump-diffusing on a hexagonal substrate layer. As the name 'Monte Carlo' suggests, this algorithm utilises random variables; while 'kinetic' implies the method simulates a dynamic, evolving system. The idea of KMC is to evolve the whole system as one 'state' instead of individual atoms. The reader can imagine the simulation drives the evolution of a trajectory in the configuration space at each timestep. At each timestep, a new configuration of atoms is generated and their positions are updated for the next calculation.

Specifically, we will discuss how random variables are used to de-

cide the future configuration, and by 'state' we mean a certain configuration of atoms. Relative to the current state, there are configurations possible to evolve into. The probability of each state is associated with and proportional to the rate constant k_i. We then align all the k_i together into a line with length $k_t = \sum k_i$. The random pointer r will point at a point on the line, whichever state is pointed that will be the future configuration. Figure 1 provides a visual illustration of this process.

The more careful readers might wonder if the rate constant is dependent on the preceding events, *i.e.* the momentum of the atoms have a preferred direction. This would be the case for weakly corrugated adsorbates and the atoms undergoing ballistic motion. However, in a low temperature and thermally equilibrated system, the intervals between jumps are often long and the atoms will have 'forgotten' where they came from. The rate constants are then independent of previous states and only dependent on the current timestep. This essentially makes the simulation a Markov Chain for those interested in the mathematical theory. In addition, by low temperature, we mean the thermal energy (k_BT, where k_B is the Boltzmann constant and T is the temperature) of the atoms is much lower than the energy barrier (E_b), *i.e.* $k_BT \ll E_b$. Physically, consider a thermally excited atom in a potential well, its random motion will have eliminated traces of previous motions after some time.

Hence, the simulation carries out a number of runs until the system is in thermal equilibrium before documenting the trajectories and carry out the analysis required for the experiment. The above describes a very simplified version of the KMC algorithm, and the actual implementation, including the codes provided to the reader, involves algorithm outside the scope of this tutorial, *e.g.* Metropolis algorithm. In the next section, we will describe the physical diffusion system we will be simulating.

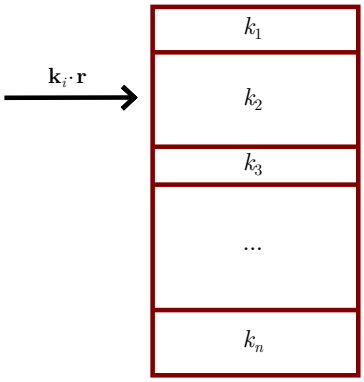

Figure 1 | Pointer r chooses the future event, each k_i represents a future state and its length is proportional to the probability of each state.

MECHANISM OF H_2O DIFFUSING ON BI_2TE_3

The codes provided to the reader and used in this tutorial are part of the KMC programme made to simulate water molecules diffusing on the surface of Bi_2Te_3 lattice [2]. The water molecules interact with each other through their dipole moment μ. The physics involving dipole moment is assumed from the reader's first-year physics. However, we will state the key equations below. Firstly, the electric potential ϕ generated by a molecule with dipole moment μ is

$$\phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\frac{\mu \cdot \hat{\mathbf{r}}}{r^2},$$

where $\mathbf{r}$ is the displacement vector, and ϵ_0 is the permittivity. Furthermore, the electric field generated by the dipole moment can be found using $\mathbf{E} = -\nabla\phi$. These equations will be useful in finding the potential energy surface (PES) of the adsorbate-substrate system.

The hopping rate of an adsorbate (water molecule) is associated with its thermal energy (temperature) and the potential barrier between grid sites. For such thermally equilibrated atoms at low energy and mostly trapped in grid sites, the hopping rate can be found using the Arrhenius relationship, [3]

$$\text{rate} = v_0 e^{-E_b/k_B T},$$

where v_0 is the rate of escape attempt and E_b, k_B, T are defined above. In a system where the jump diffusion of multiple sites is possible, *e.g.* the water molecule has 63% chance of making a one-grid jump and 37% for a two-grid jump, v_0 is a product of such rate, rate of the number of attempt ν, and the length of the timestep δt. Hence, the jumping probability p can be written as

$$p = (\beta_i \nu \delta t) e^{-E_b/k_B T},$$

where β_i is the probability of i^{th} grid jump. The grid-jumping rate β is derived empirically through experiments and is provided in the code as `jump_dist`. In the following section, we will discuss how to calculate the RDF on a 2D surface, which will be helpful to the tasks.

CALCULATING THE RDF

The RDF defines the probability of finding a particle at distance r away given there is a particle at the origin, and it is normalised by the bulk density. Since the probability of finding a particle at distance r is proportional to the number density at r, we have

$$\rho_0 g(r) = \frac{\delta n}{\text{shell volume}},$$

where dn is the number of atoms in the shell, and ρ_0 is the bulk density. In 3D space, the shell volume is the edge of a sphere; however, in 2D space, the shell becomes a ring. Hence, the shell volume in our simulation is $2\pi r \delta r$. Therefore,

$$g(r) = \frac{1}{2\pi r \rho_0} \frac{\delta n}{dr}.$$

We can find $g(r)$ by calculating the distance between every pair of particles, and bin them into different distance using a histogram algorithm. A more detailed description of the RDF can be found in the Theory Handbook under the radial distribution function chapter. Figure 2 shows an example of the RDF generated by this simulation.

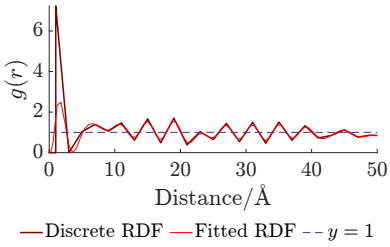

Figure 2 | The plot of both the discrete RDF and fitted RDF. The RDF approaches to 1 when r is large, as the RDF is normalised by the bulk density.

SUMMARY

- The KMC simulation evolves the whole system as a single state in each timestep.
- The evolution is dependent on the rate constant between the current state and the future-possible state.
- The rate constant is independent of preceding event due to long intervals between jump.
- The long interval is due to the low jumping rate caused by low temperature relative to energy barrier and the system thermally equilibrated.
- The adsorbates (water molecules) interact through dipole-dipole interaction.
- The hopping rate derived using the Arrhenius relationship is $p = (\beta_i \nu \delta t) e^{-E_b/k_B T}$.

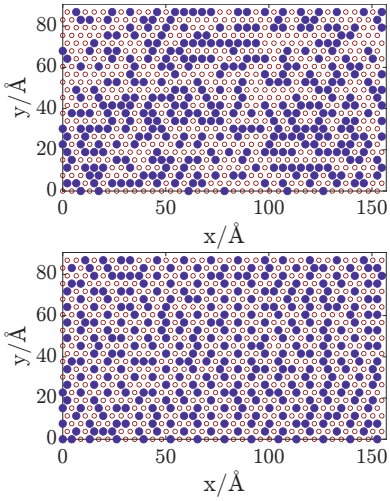

Figure 3 | The figure shows the final configurations of two jump-diffusing Monte Carlo simulations. In both simulations, the coverage is 0.4ML, and the dipole constant is 4 Debye. The temperature in the first one is $10K$, and the second is $150K$. The lower temperature simulation shows a more quasi-hexagonally ordered final configuration while the higher temperature adsorbates distribute more randomly.

TASK

1. Use the code provided, run the file `RunMCloops` using the following initial condition,
   - Coverage = 0.4 ML,
   - Dipole constant = 4,
   - Effective neighbours = 10,
   - Temperature = 150 K.

   The setting for the initial conditions can be found in the `RunMCloops` and `MC_run_loop`. The appendix includes a description of how the simulation files are constructed, the reader could find it helpful to read the appendix before attempting the tasks.
2. Plot the configuration of the adsorbate sites at the last timestep of the simulation. Add your codes to `MC_run_loop`.
3. Repeat above procedure with lower temperature. Do you notice any major differences? Can you think of a reason why? Think about what affects the jump diffusion of the adsorbates. Figure 3 shows an example of the output.
4. Open a new MatLab file and write a function `MC_rdf_single_step` that calculates the RDF at a single time step using the simulation data.
5. Write another function that calculates the average RDF over several timesteps (*e.g.* 1000) using the function you have written in task 4.
6. Plot all the results in a single figure as shown in figure 2.

TIPS

1. In task 3, there are two layers of plots in each figure, one is the grid sites and the other is the adsorbate sites. Think about how to plot each of them.
2. Calculating the average RDF can be a time consuming process, adding a timer that displays what percentage of the calculation is finished could help.

REFERENCES

1. A.F. Voter. *Introduction to the kinetic monte carlo method*. In Kurt E.Sickafus, Eugene A. Kotomin, and Blas P. Uberuaga, *Radiation Effects in Solids*, pp1-23 2007.
2. A. Tamtögl KiMCaSD - Kinetic Monte Carlo for Surface Diffusion. 07/11/2019 Zenodo. http://doi.org/10.5281/zenodo.3531646
3. A.P. Jardine, H. Hedgeland, G. Alexandrowicz, W. Allison, and J. Ellis. *Prog. Surf, Sci.*, 84:323379, 11 2009.

APPENDIX

STRUCTURE OF THE SIMULATION

The simulation is composed of several MatLab files, each with their own purposes. In the following, we will explain the functions of each file, and finally end the section with an overview of the structure between the files.

- `RunMCloops` - Define initial conditions and initialise the simulation.
- `MC_run_loop` - The main codes that decide how the simulation works, and what figures to produce, etc. This includes the main loop of the simulation.
- `MC_set_grid` - Initialise the hexagonal substrate layer.
- `MC_neighbour` - Calculates the possible locations that each adsorbates could jump diffuse to.
- `MC_init_adsorbates` - Creates initial configuration of adsorbates.
- `MC_reset_potential` - Calculates the potential surface of a given configuration of adsorbates.
- `MC_adsorbate_move` - Calculates the configuration of adsorbates in the next time step. Also renews the potential surface of the next time step.
- `calculate_ISFs` - Calculates the intermediate scattering function (ISF) after the simulation is finished.
- `fit_ISFs` - Analyse the ISFs calculated using the previous file. Includes the calculation of dephasing rate, and produces figures such as ISF against spin-echo time and dephasing rate against momentum transfer.

When running the simulation, `RunMCloops` defines the values of the initial conditions, *e.g.* temperature, coverage. These information are then fed to `MC_run_loop`, which initialises the substrate layer with `MC_set_grid` and the adsorbates with `MC_init_adsorbates`. The simulation then calculates the current potential surface with `MC_reset_potential` and the possible jump locations with `MC_neighbour` according to the initial conditions. Depending on the setting, the simulation might run until the system settles down, *i.e.* reaches thermal equilibrium. In the main loop, the locations of the adsorbates are updated using `MC_adsorbate_move` in each time step for a fix number of runs. The tra-

jectories of each adosrbates are recorded and used to calculate the ISFs using `calculate_ISFs` when the main loop ends. `fit_ISFs` analyses the result and generate different figures depending on the settings.

FUNCTIONS IN THE SIMULATION

In this section, we will take a more detailed look into the mechanism in each function. The reader does not need to understand every line of code in the simulation to use it; however, some degree of understanding will help the reader adjust the code to their purpose of use. This section is written with the aim to easing the difficulty of reading the code for the first time in mind, so the reader is encouraged to read this section along with the codes provided.

RunMCloops

As mentioned in the previous section, this function defines the initial conditions and call upon the main part of the simulation. We will define the initial variables as the following.

- `cov` - Coverage, the ratio between the number of adsorbates and substrate atoms.
- `dipole_const` - Electric dipole constant of each adsorbates, this drives the interaction between adsorbates and is measured in Debye.
- `pot_dis_num` - Effective potential neighbors, the number of neighbors that the dipole moment of an adsorbate affects.
- `numruns` - Number of simulations, note that this is different from the number of timesteps.

After the initialisation, the function calls upon `MC_run_loop`, which is the main part of the simulation.

MC_run_loop

The first section initialises the variables and the 'plotflags' that controls whether to produce several figures, including video of the simulation. The reader might find that turning off the video plotflag accelerates the simulation. `self_run_mode` controls whether to self run the simulation. The simulation self runs to stablise the system before the main simulation starts as discussed previously.

The the main loop of the simulation simulates the jump diffusion over each time step and records the trajectories. The trajectories are recorded after each time step. The trajectories will be used to calculate the ISFs and dephasing rate after the main loop, which are essential in understanding the system. Note the trajectories are separated into

two arrays, `outputx` and `outputy`. The codes then carry out the ISF calculation and analyse them using the trajectories we just simulated.

The following sections will discuss other user-defined functions. This introductory section should help the reader better understand the results produced by the functions, and help the reader write new functions suited to the purpose or their simulation. These information can also be found in the comments of the code.

MC_set_grid

`MC_set_grid` defines the grid sites of the simulation. We will define what each variables mean, hopefully shed light upon the usage of this function. If the reader is interested in the mechanism behind the code, the commented file should provide plentiful information.

Input

- nx - number of grid points in x direction.
- ny - number of grid points in y direction.
- plotflag - whether to plot the grid points.

Output

- `grid_info` - nx x ny x 4 matrix, where the third dimension is
  1. Index of the grid site.
  2. x coordinate of the grid site.
  3. y coordinate of the grid site.
  4. The potential of the grid site.
- `grid_info_periodic` - Same as `grid_info`, except the super-cell has been periodically repeated, in order to account for the boundary condition.
- `edge_x` - The x coordinate of the edge of the super-cell.
- `edge_y` - The y coordinate of the edge of the super-cell.

MC_neighbour

This simulation only consider at most 3-grid jump in each diffusion time step. The probability of each jump is found experimentally and depends on the chemical components of substrates and adsorbates.

Input

- `grid_info` - calculated in `MC_set_grid`.
- `grid_info_periodic` - same as above.

Output

- `jumpvector` - nx x ny x 3 matrix, contains the coordinates and norm of the vector of the jump step.
- `first_nearest_pair` - nx x ny x 3 matrix, contains the nearest grid sites from each point.
- `second_nearest_pair` - nx x ny x 3 matrix, contains the second nearest grid sites from each point.
- `third_nearest_pair` - nx x ny x 3 matrix, contains the third nearest grid sites from each point.
- `neighbour_info` - nx x ny x 3 matrix, contains all grid sites within effective distance. The third dimension contains the indices and coordinates of the sites.

MC_init_adsorbates

From the coverage of adsorbates and grid sites, we can randomly distribute adsorbates onto the substrate layer, and plot the configuration of the adsorbates.

Input

- `grid_info` - calculated in `MC_set_grid`.
- `plotflag` - plots initial configuration if true.

Output

`atoms` - the coordinates of the adsorbates. This will be repeatedly updated in the simulation. The cumulative result will be recorded and processed to produce the trajectory of the adsorbates.

MC_reset_potential

This functions calculates potential surface from given adsorbate configuration. Figure 4 shows an example of the potential plot.

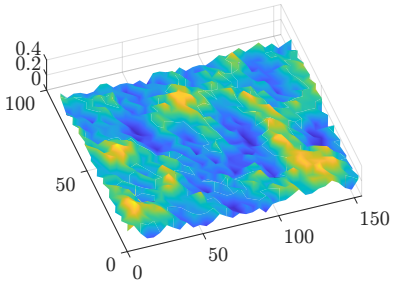

Figure 4 | The initial potential surface of the substrate-adsorbate system.

Input

- `current_potential` - the current potential surface.
- `atoms` - the current configuration of the adsorbates.
- `plotflag` - plots potential surface if true.
- `grid_info` - calculated in `MC_set_grid`.
- `neighbour_info` - calculated in `MC_neighbour`.
- `fitting_input` - an array that includes dipole constant, attempt jump rate, etc. The function only uses the dipole constant.

Output

`new_potential` - new potential energy at each site.

MC_adsorbate_move

Takes all the information calculated above, calculate the configuration of the adsorbates in the next time step.

Input

`grid_info`, `neighbour_info`, `jumpvector`, `atoms`, `first_nearest_pair`, `second_nearest_pair`, `third_nearest_pair`, `potential`, `fitting_input`. All have been calculated using above funcitons.

Output

- `atoms_new` - new adsorbates configuration.
- `potential_new` - new potential energy at each site, calculated calling `MC_reset_potential`.

calculate_ISFs

This function calculates the ISF from the simulated trajectories. Details of the intermediate scattering function can be found in the scattering function chapter of the Theory Handbook. If the reader has completed the ISF tasks in the previous MD and MC tutorial, these codes should appear familiar. The only difference would be that this is a many-particle system, and the calculation of the ISF has been adjusted accordingly by summing over all adsorbates when calculating the scattered amplitude.

Input

- dK - momentum transfer, user-defined before calculation to suit the range of interest.
- time - simulation time.
- `traj_x`, `traj_y` - x and y components of the trajectory.
- numat - number of atoms.

Ouput

- ISF - ISF values at each dK.

fit_ISFs

After deriving the ISF, we fit the ISF against the exponential form $Ae^{-\alpha x} + c$, where A is the amplitude and α is the dephasing rate. The

code then generate plots of the dephasing rate against momentum transfer.

Input

- dK - momentum transfer.
- time - simulation time.
- ISF - calculated above.
- `cut_off` - time after which the exponential is fitted, if not provided then the fitting starts at $t = 0$.

Ouput

- `analysis` - array that contains the values of A, α, c, and the range of momentum transfer.

The code includes comments that can help the reader understand the detail mechanism of each calculation, which is not required to complete the task.

SOLUTION

INTERACTIVE MC: MAIN TASK

Task 1: Question

Run the simulation using the following initial condition,

- Coverage = 0.4 ML
- Dipole constant = 4
- Effective neighbours = 10
- Temperature = 150 K.

Task 1: Solution

Change each respective initial condition in `RunMCloops` as follows.

```
    cov=0.12; %0.04 ML coverage
    dipole_const=4; %constant for the dipole forces
    pot_dis_num=10; %number of neighbours to be included in the
        dipole forces
```

Change each respective initial condition in `MC_run_loop` as follows.

```
    ann=4.36; % surface lattice spacing (unit: Angstrom)
    E_diff=0.035; % energy barrier between two sites in eV
    jump_dist = [0.63,0.37]; %probaility of 1-nn, 2-nn and 3-nn
        jumps.
    coverage=cov; % if 0 < coverage <1, calculate the real
        number according to coverage, if >1 use this number as
        number of adsorbates
    potential_eff_dis=pot_dis_num*ann;
    temperature=100; %temperature in Kelvin
    dipole_constant=dipole_const; %positive - repulsive interact
        , negative - attractive interact
    %constants
    kb=8.6173324e-5; % Boltzmann constant (unit: eV/K);
```

Run `RunMCloops` in the command line.

Task 2: Question

Plot the configuration of the adsorbate sites at the end of the simulation.

Task 2: Solution

At the end of the main loop in `MC_run_loop`, add the following lines to plot the configuration using the trajectories simulated.

```
    %new figure environment
    figure

    %plotting two layers, hold on
    hold on
    box on

    %grid sites layer
    plot(grid_info(:,:,2), grid_info(:,:,3), 'o')
    %adsorbates layer
    plot(trajx(:,end),trajy(:,end), 'o', 'markersize', 10);
```

Run `RunMCloops` in the command line.

Task 3: Question

Repeat above procedure with lower temperature. Do you notice any major differences? Can you think of a reason why? Think about what affects the jump diffusion of the adsorbates. Figure 3 shows an example of the output.

Task 3: Solution

Repeat the solution of task 2, but change the temperature at the start of `MC_run_loop`.

The reader might notice that the configuration of the adsorbates appear more ordered at higher temperature. In fact, the potential barriers of the substrate layer is insignificant compared to the thermal energy of adsorbates at high temperature. Hence, the substrate layer appears uncorrugated to the adsorbates and it becomes easy for the adsorbates to maintain an optimal distance away from each other, forming a quasi-hexagonal structure.

At lower temperature, the adsorbates becomes sluggish and like to stay in their grid sites without moving. As the initial configuration was random, there could be clumps of adsorbates in one area and this feature is preserved at the end of the simulation.

Task 4: Question

Open a new MatLab file and write a function `MC_rdf_single_step` that calculates the RDF at a single time step using the simulation data.

Task 4: Solution

Define the function `MC_rdf_single_step` that calculates the RDF at each time step.

```
function [rdf] = MC_rdf_single_step(traj_x, traj_y, numat)
    %%calculates rdf at each time step
    %Inputs-----------------------------------------------
    %trajectory of adsorbates at a single timestep
    %given in coordinates traj_x and traj_y
    %traj_x and traj_y has one dimension
    %    dim 1 - particle index
    %Outputs----------------------------------------------
    % rdf - the discrete RDF function
```

Initialise the variables, including two global variables defined in the `MC_run_loop` file.

```
    %%initialise variables
    %global-----------------------------
    %coverage - particle density
    %ann - site width
    global coverage ann
    %number of bins for histogram count
    bin_num = 100;
    %array stores distances between pairs of particles
    distances = [];
```

Go through every particle pairs and calculate the distances between them. `distance_temp` temporarily stores the distance calculated. The code then appends the value to the array `distances`.

```
%%main loop
%goes through each pair of particles
for nn = 1:numat

    for mm = 1:nn
            %calculates distance between two particles
            distance_temp = ((traj_x(nn)-traj_x(mm)).^2+(
                traj_y(nn)-traj_y(mm)).^2).^0.5;

            %append distance to distances array
            distances = [distances distance_temp];
    end
end
```

Calculate the histogram counts.

```
%%use the histcounts function to bin the distances
%edges - the edge of each bin
%h - values in each bin
[h,edges] = histcounts(distances, bin_num);
```

To find the discrete function we find the mid points of the bins and their respective values. Calculate the bulk density, which we will use to normalise the RDF.

```
%calculate width of each bin
width = edges(2)-edges(1);

%interval - the mid points of each bin
interval = edges(1:end-1)+width/2*ones(1,size(edges,2)-1);

%bulk_density - number of atoms per unit area
bulk_density = coverage*(ann^2);
```

The normalising factor varies for different r, `normalising_factor` stores all of them into an array, which should have the same dimension as `h`.

```
43      %%normalise counts
44      %the shell is 2D, uses 2D shell width, i.e. 2*pi*r*dr
45      %normalising_factor is an array for different radius
46      normalising_factor = 2*pi*bulk_density*interval;
```

Normalise RDF. From theory, we know the RDF should equals to zero from $r = 0$ to at least the width of grid, the commented code adds zeros points in those range.

```
47      rdf = [interval ; h./normalising_factor];
48      %adding the zeros for the fitting curve to drop to zero
            before the first value
49      %rdf = [ [linspace(0,2*ann,100);zeros(1,100)] , rdf];
50  end
```

Task 5: Question

Write another function that calculates the average RDF over several timesteps (*e.g.* 1000) using the function you have written in task 4.

Task 5: Solution

First, define the function `MC_rdf`.

```
1       %%create new file MC_rdf
2
3       function [rdf, fit_rdf] = MC_rdf(traj_x, traj_y, numat,
            plotflag_rdf)
4           %%main function
5           %Inputs------------------------------------------------
6           %trajectory of adsorbates given in coordinates traj_x
                and traj_y
7           %traj_x and traj_y has two dimensions
8           %    dim 1 - particle index
9           %    dim 2 - time step
10          %Outputs-----------------------------------------------
11          % rdf - the discrete RDF function
12          % fit_rdf - curve fitted using dicrete rdf
```

Initialise variables, including `rdf_many` that stores rdf of different timesteps.

```
13          %%initilise variables
14          %the number of time steps we average over
15          time_length = 1000;
16
17          %this can be a time consuming process
18          %timer shows the percentage of the code ran
19          timer = 10;
20
21          %the discrete rdf of each time step is a page of
                rdf_many
22          rdf_many = [];
```

The main loop calculates the rdf at each timestep, and updates the timer.

```
        %%main loop
        %loop over timesteps
        for time = 1:time_length
            %MC_rdf_single_step calculates the discrete rdf at
                each time step
            %The reader could have written it all in a single
                function
            %The function is separated for readibility
            rdf_many(:,:,time) = MC_rdf_single_step(traj_x(:,
                time), traj_y(:,time),numat);

            %displays the percentage of code ran
            if time/time_length > timer/100
                disp(['Finished ' num2str(timer) '% of RDF
                    calculation'])
                timer = timer + 10;
            end
        end
```

Finally, calculate the average over the timesteps.

```
        %%taking average over time steps
        rdf = mean(rdf_many,3);
```

Having calculated the discrete RDF, we can calculate the fitted curve using MatLab's Curve Fitting Toolbox.

```
        %%fitting procedure
        [dataX, dataY] = prepareCurveData(rdf(1,:),rdf(2,:));

        %'smoothingspline' as we do not know the analytic form
            of rdf
        fit_rdf = fit(dataX,dataY,'smoothingspline');
```

The `if` statement calls the plot function if `plotflag_rdf` is on.

```
        %%plotting
        if plotflag_rdf
            %plot function
            plot_rdf(rdf,fit_rdf)
        end
    end
```

Task 6: Question

Plot all the results in a single figure as shown in figure 2

Task 6: Solution

Next, we will write a function that plots the RDF, fitted RDF, and the asymptote. First, define the function and start a new figure environment. The `rdf` and `fit_rdf` are calculated in `MC_rdf`.

```
function plot_rdf(rdf, fit_rdf)
    %figure environment
figure
hold on
```

Plot the discrete and fitted RDF.

```
5       %plot the dicrete rdf
6       plot(rdf(1,:),rdf(2,:),'-')
7
8       %plot the fitted rdf
9       plot(fit_rdf)
```

Plot the asymptote $y = 1$. The RDF approaches 1 when r is large because the macroscopic property takes over. The Theory Handbook outlines a more detailed explanation.

```
10      %rdf asymptote to 1 at large r, plot dashed line of y=1
11      x=0:500;
12      plot(x,ones(1,size(x,2)),'--')
13      xlim([0 20])
14      ylim([0 inf])
```

Set legend and axis labels.

```
15      %set axes
16      xlabel('Distance/\AA', 'Interpreter','latex')
17      ylabel('$g(r)$','interpreter','latex')
18
19      %set legend
20      legend('Discrete RDF','Fitted RDF','$y=1$', 'interpreter','
            latex')
21  end
```