ASSIGNMENT 5: SOLUTION

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

Written in XƎLATEX

MONTE CARLO

INTRODUCTION

In this tutorial, we will write a simple Monte Carlo simulation for single-jump diffusion of an adsorbate on a square lattice. This is the precursor to more advanced methods of modelling surface diffusion, such as molecular dynamics, while offering a testable model for the behaviour of a particle on a surface.

THEORETICAL BACKGROUND

Consider a particle diffusing between sites on a surface. In the context of single-jump diffusion on a square lattice, it is assumed that only jumps to adjacent sites are possible. We assume that jumping to each of the four neighbouring sites is equally likely, and governed by a constant-probability Arrhenius equation. This is the essence of the entire problem - in each step, the particle randomly jumps to an adjacent site. This type of model is known as a "Monte Carlo" model of surface diffusion

In principle, it may be possible for a given adsorbate to jump over more than one lattice site in the time taken for a helium spin-echo measurement to be made, so it is of interest to also model such systems. Intuitively, the probability of jumping several lattice sites in a given timestep should be lower than jumping a single site. Implementing this possibility is explored in the extension questions.

TASK

1. Construct a Monte Carlo simulation of idealised hopping. Generate the trajectory of a diffusing adsorbate as a function of time. The surface is modelled as a discrete square lattice, with a lattice parameter of 2.5 Å.
2. For an isolated particle, plot the position of the particle along each axis as a function of time. Plot the two-dimensional trajectory of the particle.

3. Repeat task 1 for a triangular lattice (*i.e.* a lattice of tesselated equilateral triangles).
4. Repeat task 2 for a hexagonal lattice. Why is this more difficult?

Tips

1. It it convenient to generate three vectors, x, y and t, describing the positions of the particle, and their corresponding time variable. We assume the particle starts at the origin. For each timestep, we must decide whether or not the particle jumps, using the probability p of hopping (*e.g.* $p \approx 0.01$). This can be done by sampling a value h from the Bernoulli Distribution Ber(p). If the value of this sample h is 1, the particle hops. If the sample h is 0, then it does not. We also need to know which direction it will jump in. This can be done by taking a random integer from 1 to 4.

EXTENSION

1. Write a subclass that allows for double jumps on the square lattice. Intuitively the probability of performing multiple jumps is smaller than single jumps, but experiment with varying the probability distribution.
2. Construct a class that allows for jumps of infinite length with a Gaussian distribution and standard deviation of 1 jump.

ISF FROM MONTE CARLO

INTRODUCTION

In this tutorial, we will use the Monte Carlo trajectories, determined in the chapter above, to calculate the intermediate scattering function (ISF) for quasi-elastic helium atom scattering (QHAS). We will analyse the behaviour of the ISF for different parallel momentum transfers.

THEORETICAL BACKGROUND

ISF as Autocorrelation

The knowledge of the trajectories of adsorbed atoms on a surface can allow us to model the scattering of incident helium atoms from the sample. Fortunately, this is the information we obtained in the first task! In this section, we will discuss the theory related to the intermediate scattering function for our system, allowing us to make sense of this data. Given we are considering quasi-elastic scattering, momentum can be transferred from the beam to phonons in the lattice; consequently the parallel momentum transfer $\Delta\mathbf{K}$ can take many values. In typical experiments, the parallel momentum transfer ΔK ranges from $0\,\text{Å}^{-1}$ to $4\,\text{Å}^{-1}$, and it can point along any direction in the surface.

How do we find the amplitude of the scattered beams? Using a simple kinematic scattering model, the scattered amplitude A is given by:

$$A = FS,$$

where F is a "form factor" for scattering; it determines the distribution of scattered intensity from a given scattering centre. S is the structure factor for the surface. In general, we do not know the form of the atomic factor F - we can assume it is 1. It is straightforward to discuss the surface, but what about the inside of the solid? We assume the solid's internal structure has little effect on scattering. Thus it is not taken into consideration for our calculations.

What is the structure factor S? It is a quantity that encodes the phase change due to the relative positions of particles. For slowmoving adsorbates, it is given by:

$$S = e^{-i\Delta\mathbf{K}\cdot\mathbf{R}_a(t)},$$

where $\mathbf{R}_a$ is the trajectory of a particle. Thus for any trajectory $\mathbf{R}_a(t)$, the expression for the complex scattered amplitude A reduces to

$$A(\Delta\mathbf{K}, t) = e^{-i\Delta\mathbf{K}\cdot\mathbf{R}_a(t)}.$$

What use is this formalism in experiment? The observable quantity in spin-echo experiments is the intermediate scattering function (ISF). The ISF can be described, among other things, as the autocorrelation of the time-dependent scattering amplitude $A(t)$. What does this mean? Given that the autocorrelation of a function is the convolution of the function with its complex-conjugate, with time as a variable, the convolution theorem suggests that we can write the ISF $I(\Delta\mathbf{K}, t)$ as:

$$\begin{aligned} I(\Delta\mathbf{K}, t) &= A(\Delta\mathbf{K}, t) * A(\Delta\mathbf{K}, t)^*; \\ &= \mathcal{F}^{-1}\left[\mathcal{F}[A(t)] \cdot \mathcal{F}[A(t)^*]\right]; \\ &= \mathcal{F}^{-1}\left[\mathcal{F}[A(t)] \cdot \mathcal{F}[A(-t)]^*\right]. \end{aligned}$$

Can we simplify this further? The complex scattered amplitude $A(t)$ is symmetric about $t = 0$, as the motion of the particles is random. Thus, the expression for the ISF above reduces to

$$I(\Delta\mathbf{K}, t) = \mathcal{F}^{-1}\left[|\mathcal{F}[A(\Delta\mathbf{K}, t)]|^2\right].$$

This formulation can equivalently be obtained from the inverse temporal Fourier transform of the dynamic structure factor.

Statistical Approach

The ISF provides a complete statistical description of adsorbate motion on a surface. However, there is usually more information contained in this data than can be feasibly extracted. How do we get around this issue? In QHAS experiments, we fit the ISF with an exponentially decaying function; the exponential prefactor α in this function is called the "dephasing rate". The behaviour of the prefactor α as the parallel momentum transfer ΔK varies is drastically different for distinct types of motion on the surface.

A useful description of the motion of a particle is the van Hove pair correlation function. We define (classically) the van Hove pair correlation function $G(\mathbf{R}, t)$ as the probability of finding a particle at the position $\mathbf{R}$ at a time t, given there was a particle at the origin at $t = 0$. Since we are only considering a single isolated adsorbate, we can write the van Hove pair correlation function as the trajectory of the particle (as the probability of finding it at that point is 1), as

$$G(\mathbf{R}, t) = G_s(\mathbf{R}, t).$$

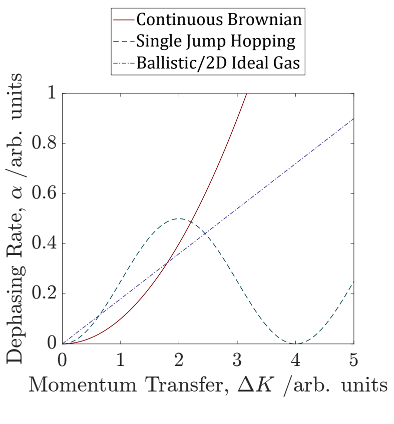

Figure 1 | Variation of the measured dephasing rate with parallel momentum transfer for simple diffusion models.

The position of the particle as a function of time depends on the diffusive regime it experiences while diffusing on the surface. Summarily, these can be generally classified as continuous Brownian motion, single jump hopping, and “ballistic” motion. These will be explored individually below.

- The simplest model for a diffusing particle is continuous, random, Brownian motion. For this regime, we expect the form of the van Hove pair correlation function G_s to be:

  $$G_s(\mathbf{R}, t) = \frac{1}{4\pi D|t|} \exp\left(-\frac{R^2}{4D|t|}\right),$$

  where D is a constant. By Fourier transforming, this yields the intermediate scattering function (ISF) as

  $$I_s(\Delta\mathbf{K}, t) = \exp(-\Delta K^2 D|t|).$$

  From this, the the dephasing rate α evidently obeys the relation

  $$\alpha(\Delta\mathbf{K}) \propto \Delta K^2$$

  (*i.e.* it is proportional to the modulus squared of the parallel momentum transfer). This behaviour occurs in the limit of large length-scales and temperatures, or measurements of sufficiently small momentum transfers ΔK.
- At sufficiently low temperatures, adsorbate motion becomes dominated by the periodic arrangement of substrate atoms; we expect to see the adsorbates undergoing discrete jumps as a result. This follows the rate equation for instantaneous jumps:

  $$\frac{\partial G_s(\mathbf{R}, t)}{\partial t} = \sum_{\mathbf{j}} v_{\mathbf{j}}[G_s(\mathbf{R} + \mathbf{j}, t) - G_s(\mathbf{R}, t)],$$

  where $\mathbf{j}$ are the allowed jump vectors on the lattice. This can be solved to give the dephasing rate as

  $$\alpha(\Delta\mathbf{K}) = 2\sum_{\mathbf{j}} v_{\mathbf{j}} \sin^2\left(\frac{\Delta\mathbf{K}\cdot\mathbf{j}}{2}\right).$$

  Can we make any remarks on this equation? This sinusoidal behaviour gives zeroes at a discrete set of points; these points satisfy the condition that the parallel momentum transfer $\Delta\mathbf{K}$ is the same as any reciprocal lattice vector $\mathbf{G}$. It is reassuring to see that the dephasing rate is periodic with position, especially in a way that respects the periodicity of the lattice.
  What happens if the adsorbate jumps further than one spacing in a time interval? We would need to include more components in the series for α. It is trivial that in the limit of small ΔK we recover the quadratic dependence.

- There is one more case to consider. When the substrate corrugation is small compared to the energy of the adsorbate, then “ballistic” motion is expected for the adsorbate. This is especially clear for small length scales and time scales. The van Hove pair correlation function is then predicted by statistical mechanics to be:

  $$G_s(\mathbf{R}, t) = \frac{1}{\pi(v_0 t)^2} \exp\left(-\frac{R^2}{(v_0 t)^2}\right),$$

  where the temperature is related to the root mean square velocity of the particles through the equipartition theorem as:

  $$k_B T_s = \frac{1}{2} m v_0^2,$$

  where v_0 is the root mean square velocity of the adsorbates. This assumes that there are no additional quadratic potential energies acting as degrees of freedom (such as chemical bonds). Following through with calculating the intermediate scattering function through the Fourier transform suggests that the dephasing rate α can be written as

  $$\alpha(\Delta\mathbf{K}) = \sqrt{\ln 2} v_0 \Delta K.$$

  In this case, the dephasing rate α is proportional to the magnitude of the parallel momentum transfer ΔK.

CURVE FITTING

The Matlab `fit()` function uses iterative optimisation to calculate the coefficients for a fitted function to a set of data. Providing `fit()` with a known set of data points, along with a function to fit it to, MATLAB can produce a fitted curve. It can also provide the calculated coefficients in the fitted curve.

```
% Setup the x axis and a function
x = linspace(-2,2)';
y = cos(x);


% Define the type of function to be fitted
g = fittype('1-a*x^2');


% This command fits the data to the curve
% Initial guess for parameter a is the final argument
fit_g = fit(x, y, g, 'startpoint', [0.4]);


% Returns vector of fit coefficient values
a_est = coeffvalues(fit_g);


% Plot and save
```

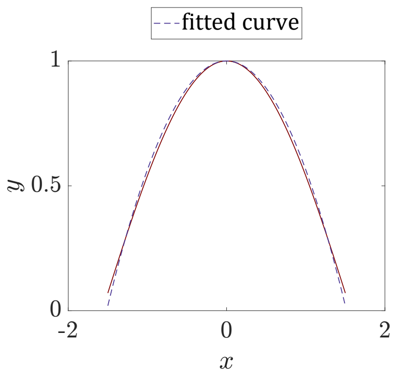

Figure 2 | A fitted curve matches closely to a set of data.

```
plot(x, y);
hold on;
plot(fit_g);
```

which produces the following fit, with $a_{\text{est.}} = 0.4704$:

TASK

1. Using the trajectory $\mathbf{R}(t)$ of an isolated adsorbate on a square lattice, calculate and plot the scattered amplitude $A(\Delta\mathbf{K}, t)$ as a function of time for a particular value of the parallel momentum transfer ΔK (use $2\,\text{Å}^{-1}$ as the parallel momentum transfer).
2. Using the analytic expression for the ISF in terms of the complex scattered amplitude $A(\Delta\mathbf{K}, t)$, calculate the ISF for the same value of ΔK as used above. Note that the timebase of the ISF is half that of the amplitude. Why is this?
   (a) Redefine your ISF to only include the first time-base, then plot your ISF against time for the new time-base. Your plot should look like an exponential decay, descending into noise with increasing time.
   (b) Use the matlab `fit()` function to fit

$$I(\Delta\mathbf{K}, t) = e^{-\alpha(\Delta\mathbf{K})t}$$

   to the calculated ISF, and plot the fitted curve over the simulation curve to check that it fits theoretical prediction.
3. Try varying the following parameters and observe how they affect the ISF:
   - Lattice parameter a
   - Parallel momentum transfer ΔK
   - Hopping probability p
   - Scattering direction $\arg(\Delta\mathbf{K})$
   - Timestep between points in the trajectory
   - Total length of the simulation
4. By iterating over the code you have already written, either with a loop or using vector operations, calculate α for a series of values of ΔK, then plot α as a function of ΔK. You should find that your plot has a similar form to the single-jump diffusion model.
5. Try plotting the same curve for triangular and hexagonal lattices. What do you notice about the new dephasing rate curves?
6. You may notice that, despite following the theoretical curve on average, the dephasing curves become significantly more noisy for lower numbers of timesteps. Why is this?

EXTENSION

1. Try plotting the dephasing rate curve for the double-jump simulation. Add higher Fourier components to your fitted curve, as suggested in the theoretical background section. Experiment with varying the probability distribution.

SOLUTIONS

MONTE CARLO: MAIN TASKS

Task 1: Question

Construct a Monte Carlo simulation of idealised hopping, and generate the trajectory of a diffusing adsorbate as a function of time. The surface is modelled as a discrete square lattice with parameter of approximately 2.5 Å. There is a finite probability of hopping to a random neighbouring site at each timestep.

Task 1: Solution

In these solutions, we jump straight to using an object-oriented approach. We first define a handle class `MCSimu`:

```
%-----------------------------------------------------------------
% define a encompassing class
%-----------------------------------------------------------------
classdef (Abstract) MCSimu < handle
```

We then define the important properties. We will store the lateral positions of the adsorbate, and the current time of the simulation, in three vectors `x`, `y` and `t`.

```
    %-------------------------------------------------------------
    % define the properties of the class
    %-------------------------------------------------------------
    properties
        % length of each timestep /microseconds
        timestep = 1;

        % Probability of a particle jumping one space in one
            timestep
        hopProb = 0.01;

        % lattice parameter /A
        lattConst = 2.5;

        % 3 vectors recording the trajectory of the particle
        x = [];
        y = [];
        t = [];

        % maximum no. timesteps allocated for the simulation,
            including t=0 state
        numSteps = 10;
    end
    %-----------------------------------------------------------
    %-----------------------------------------------------------
```

It is useful to define a few abstract methods here. While this isn't necessary, it allows Matlab to implement these methods for all subclasses of `MCSimu` before they run. It is good practice as a programmer to ensure that your classes will always function as expected, if another user extends them to a subclass. These methods are:

```
    methods(Static, Abstract)
        % abstract - samples range of integers corresponding to jump directions
        hopDirec()
    end

    methods (Abstract)
        % abstract - propagates the state of the simulation through n timesteps of length this.timestep
        nStates_loop(obj, n)
    end
```

Now we turn to writing the function to construct the Monte Carlo simulation. The key issue is how to handle the vectors `x`, `y` and `t`. In principle, it would be possible to make these resizable, and to allow simulations of any length; this is problematic for longer simulations, and so the variable `numSteps` is implemented to terminate the function after a certain number of steps. This function is defined as:

```
    methods
        %-------------------------------------------------------
        % define a function MCSimu_sqr
        %-------------------------------------------------------
        function obj = MCSimu_sqr(hopProb, lattConst, timestep, numSteps)
            % CONSTRUCTOR

            % If the correct no. of arguments are passed in...
            if nargin > 0

                % Copy passed in arguments of constructor to their corresponding properties:

                % jump probability (in any direction) at each timestep
                obj.hopProb = hopProb;

                % lattice spacing of grid /A
                obj.lattConst = lattConst;

                % length of each timestep /ps
                obj.timestep = timestep;

                % maximum number of timesteps that simulation can store
                obj.numSteps = numSteps;

                % Initialise the x,y,t vectors with max length
                    of numSteps:
                obj.x = zeros(numSteps,1);
                obj.y = zeros(numSteps,1);
                obj.t = zeros(numSteps,1);
            end

            % Otherwise, just use the default values
        end
        %--------------------------------------------------------
        %--------------------------------------------------------
```

he only remaining functions to implement are `hopOrNot()` and `plotAll()`. The former simply returns 1 with probability `hopProb` and the latter plots graphs of x vs$t$, y vs t and y vs x. These are easily implemented:

```
        %--------------------------------------------------------
        % define a function hopOrNot
        %--------------------------------------------------------
        function h = hopOrNot(obj)
            % Single Bernoulli trial => gives 1 w/. prob p:
            h = binornd(1, obj.hopProb);
        end
        %--------------------------------------------------------
        %--------------------------------------------------------



        %--------------------------------------------------------
        % define a function plotAll
        %--------------------------------------------------------
        function plotAll(obj)
            % Plot all three graphs determining the trajectory.
            figure();
            plot(obj.t, obj.x);
            xlabel('t /micro s'); ylabel('x /A');

            figure();
            plot(obj.t, obj.y);
            xlabel('t /micro s'); ylabel('y /A');

            figure();
            plot(obj.x, obj.y); axis equal;
            xlabel('x /A'); ylabel('y /A');
        end
        %--------------------------------------------------------
        %--------------------------------------------------------
    end
end
```

Having defined a parent class `MCSimu`, we can define a subclass `MCSimu_sqr` to solve our problem. Supposing we consider only single jumps, there are four possible locations to which jumps can occur; the result of this is that the function `hopDirec()` randomly samples an integer between 1 and 4.

```
classdef MCSimu_sqr < MCSimu

    methods(Static)
        %-------------------------------------------------------
        %-----------------------------------------------------
        function d = hopDirec()
            % Define: 1 up, 2 right, 3 down, 4 left
            d = randi(4);
        end
        %-----------------------------------------------------
        %-----------------------------------------------------
    end
```

For a square lattice, the hop directions coincide with the x-axis and the y-axis; in the event of a hop we can use a switch statement on `d` (equivalent to an if/else statement for `d` as 1, 2, 3, and 4). Recall that the the number of steps `n` should not exceed the constant `numSteps`, as this represents the length of our `x`, `y` and `t` vectors.

```
    methods
        %-----------------------------------------------------
        % define a function nStates_loop
        %-----------------------------------------------------
        function nStates_loop(obj, n)
        % Propagates the state of the simulation through n
            timesteps of length this.timestep

            % for n timesteps...
            for i = 2:n

                % if the particle jumps...
                if obj.hopOrNot() == 1

                    % randomly sample amongst 4 jump directions
                    d = obj.hopDirec();

                    % execute jump
                    switch d
                        % Move (x,y) -> (x,y+a)
                        case 1
                            obj.x(i) = obj.x(i-1);
                            obj.y(i) = obj.y(i-1) + obj.
                                lattConst;

                        % Move (x,y) -> (x+a,y)
                        case 2
                            obj.x(i) = obj.x(i-1) + obj.
                                lattConst;
                            obj.y(i) = obj.y(i-1);

                        % Move (x,y) -> (x,y-a)
                        case 3
                            obj.x(i) = obj.x(i-1);
                            obj.y(i) = obj.y(i-1) - obj.
                                lattConst;

                        % Move (x,y) -> (x-a,y)
                        otherwise
                            obj.x(i) = obj.x(i-1) - obj.
                                lattConst;
                            obj.y(i) = obj.y(i-1);
                    end

                % if it doesn't jump...
                else
                    % record that its position is unaltered
                    obj.x(i) = obj.x(i-1);
                    obj.y(i) = obj.y(i-1);

                % end conditions
                end

                % increment time by one timestep
                obj.t(i) = obj.t(i-1) + obj.timestep;
            end

        % end for loop
        end

    % end function
    end

% end class
end
```

This greatly simplifies our `main.m` `function`. We begin by initialising the parameters required for the simulation:

```
% Avoids a cluttered workspace
clear;

%----------------------------------------------------------------
% initialise lattice parameters
%----------------------------------------------------------------

% hop probability per timestep
p = 0.01;

% lattice parameter /A
a = 2.5;

% Number of timesteps in trajectory simulation
n_t = 100000;

% length of timesteps /micro s
dt = 1;

%----------------------------------------------------------------
%----------------------------------------------------------------

%----------------------------------------------------------------
% run MC simulation to generate trajectory
%----------------------------------------------------------------

% instantiate simulation object with max number of timesteps n_t
mcs = MCSimulation(p, a, dt, n_t);

% run n_t timesteps
mcs.nStates_loop(n_t);

% plot trajectory
mcs.plotAll();

%----------------------------------------------------------------
%----------------------------------------------------------------
```

Task 2: Question

Plot x vs t, y vs t and y vs x for an isolated particle diffusing on the surface.

Task 2: Solution

Executing the previous code produces figures similar to the figures in the margin.

Task 3: Question

Repeat task 1 for a triangular lattice (*i.e.* a lattice of tesselated equilateral triangles) by creating an "`MCSimu_tri`" subclass.

Task 3: Solution

For the (equilateral) triangular lattice we still have that each lattice site is equivalent, however now there are six possible jumps at each site:

```
%-------------------------------------------------------------
% define a subclass MCSimu_tri for the problem
%-------------------------------------------------------------
classdef MCSimu_tri < MCSimu

    methods(Static)
        function d = hopDirec()
            % Define: 1 up, 2 right, 3 down, 4 left
            d = randi(6);
        end
    end
```

The hop directions no longer coincide with the x and y axes, so some trigonometry is required to calculate the changes in `x` and `y` resulting from each hop direction. Besides that this function is exactly the same:

```
    methods
        function nStates_loop(obj, n)
            for i = 2:n
                if obj.hopOrNot() == 1
                    d = obj.hopDirec();
                    switch d
                        case 1
                            obj.x(i) = obj.x(i-1) + obj.
                                lattConst;
                            obj.y(i) = obj.y(i-1);
                        case 2
                            obj.x(i) = obj.x(i-1) + 0.5*obj.
                                lattConst;
                            obj.y(i) = obj.y(i-1) - 0.5*sqrt(3)*
                                obj.lattConst;
                        case 3
                            obj.x(i) = obj.x(i-1) - 0.5*obj.
                                lattConst;
                            obj.y(i) = obj.y(i-1) - 0.5*sqrt(3)*
                                obj.lattConst;
                        case 4
```

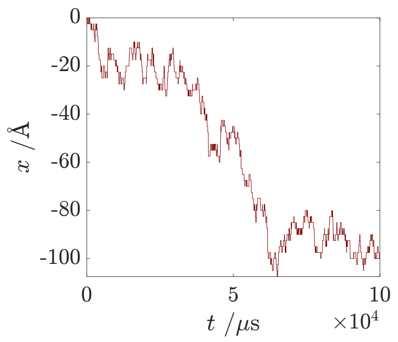

Figure 3 | Plot of the displacement in the x-direction of a single adsorbate against time on a square grid.

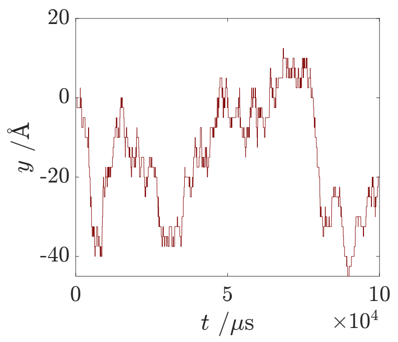

Figure 4 | Plot of the displacement in the y-direction of a single adsorbate against time on a square grid.

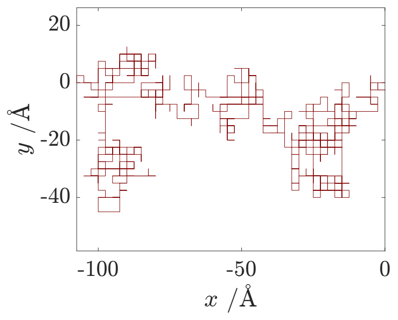

Figure 5 | Plot of the y on x trajectory of a single adsorbate on a square grid.

```
                        obj.x(i) = obj.x(i-1) - obj.
                            lattConst;
                        obj.y(i) = obj.y(i-1);
                    case 5
                        obj.x(i) = obj.x(i-1) - 0.5*obj.
                            lattConst;
                        obj.y(i) = obj.y(i-1) + 0.5*sqrt(3)*
                            obj.lattConst;
                    otherwise
                        obj.x(i) = obj.x(i-1) + 0.5*obj.
                            lattConst;
                        obj.y(i) = obj.y(i-1) + 0.5*sqrt(3)*
                            obj.lattConst;
                end
            else
                obj.y(i) = obj.y(i-1);
                obj.x(i) = obj.x(i-1);
            end
            obj.t(i) = obj.t(i-1) + obj.timestep;
        end
    end
end
end
%----------------------------------------------------------------
%----------------------------------------------------------------
```

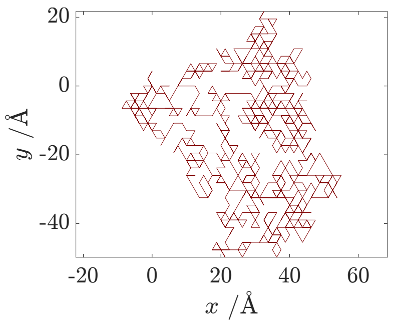

Figure 6 | Plot of the y on x trajectory of a single adsorbate on a tirangular grid.

The new class also requires a constructor, but this is the same as that of the superclass. Unfortunately Maltab doesn't allow inheritance for constructors.

All that need be changed in the main function is the subclass used. This code produces a figure similar to the one in the margin.

Task 4: Question

Repeat task 2 for a hexagonal lattice. Why is this more difficult?

Task 4: Solution

Doing the same for the hexagonal lattice is more complicated, since there are effectively two types of site (really it's a lattice with a motif of two different sites). Each time the adsorbate moves on the lattice, it switches site. However, the symmetry of this system is exploitable.

At both sites there are three jump directions, so we only need to sample three:

```
%----------------------------------------------------------------
% define a subclass MCSimu_hex for the problem
%----------------------------------------------------------------
classdef MCSimu_hex < MCSimu

    methods(Static)
        function d = hopDirec()
            % Define: 1 up, 2 right, 3 down, 4 left
            d = randi(3);
        end
    end
```

Due to the symmetry of the system we can easily switch between cases as the hop directions at each site are reflections of each other in a vertical line (x is constant). Since we switch site on each jump, we can multiply the horizontal (x) components of the jumps by $(-1)^{\mu}$, where μ is a constant that changes between 0 and 1 after each jump. Unfortunately we can't simply use the index `i` for μ since this changes even if no jump occurs. Besides this technicality, the method is largely the same:

```
    method
        function nStates_loop(obj, n)

            % defined to be 0/1 to represent the type of site we
                 are on
            siteType = 0;

            for i = 2:n
                if obj.hopOrNot() == 1
                    d = obj.hopDirec();
                    switch d
                        case 1
                            obj.x(i) = obj.x(i-1) + (-1)^
                                siteType*obj.lattConst;
                            obj.y(i) = obj.y(i-1);
                        case 2
                            obj.x(i) = obj.x(i-1) - (-1)^
                                siteType*0.5*obj.lattConst;
                            obj.y(i) = obj.y(i-1) - 0.5*sqrt(3)*
                                obj.lattConst;
                        otherwise
                            obj.x(i) = obj.x(i-1) - (-1)^
                                siteType*0.5*obj.lattConst;
                            obj.y(i) = obj.y(i-1) + 0.5*sqrt(3)*
                                obj.lattConst;
                    end

                    % awitch siteType if adsorbate moves,
                        otherwise leave it alone
                    if siteType == 0
                        siteType = 1;
                    else
                        siteType = 0;
                    end
                else
                    obj.y(i) = obj.y(i-1);
                    obj.x(i) = obj.x(i-1);
                end
                obj.t(i) = obj.t(i-1) + obj.timestep;
            end
        end
    end
end
%----------------------------------------------------------------
%----------------------------------------------------------------
```

This produces a figure similar to the figure in the margin.

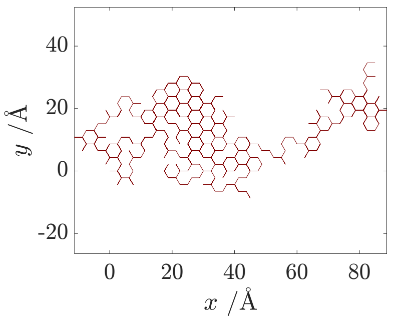

Figure 7 | Plot of the y on x of a single adsorbate on a hexagonal grid.

ISF FROM MONTE CARLO: MAIN TASKS

Task 1: Question

Using the trajectory $\mathbf{R}(t)$ of an isolated adsorbate on a square lattice, calculate and plot the scattered amplitude $A(\Delta\mathbf{K}, t)$ as a function of time for a particular value of the parallel momentum transfer ΔK (use $2\,\text{Å}^{-1}$ as the parallel momentum transfer) along a particular azimuth.

Task 1: Solution

We first run the code from the previous tutorial to generate a single trajectory. We also define a single momentum transfer modulus of interest along a given azimuth.

```
% avoids a cluttered workspace
clear;

%-----------------------------------------------------------------
% define simulation parameters
%-----------------------------------------------------------------

% hop probability
p = 0.01;

% lattice parameter /A
a = 2.5;

% number of timesteps in trajectory simulation
n_t = 1000;

% length of timesteps /micro s
dt = 1;

%-----------------------------------------------------------------
%-----------------------------------------------------------------


%-----------------------------------------------------------------
% define single momentum transfer we are interested in
%-----------------------------------------------------------------

% modulus of parallel momentum transfer /A^{-1}
mod_DK = 2;

% direction of DK /rad
arg_DK = 0;

% set of DK Vectors /A^{-1}
DK = mod_DK .* [cos(arg_DK) sin(arg_DK)]';

%-----------------------------------------------------------------
%-----------------------------------------------------------------


%-----------------------------------------------------------------
% run MC simulation to generate a single trajectory
%-----------------------------------------------------------------
```

```

% instantiate square simulation object
mcs = MCSimu_sqr(p, a, dt, n_t);

% run simulation
mcs.nStates_loop(n_t);

% extract trajectory
x = mcs.x; y = mcs.y; t = mcs.t;
%------------------------------------------------------------------
%------------------------------------------------------------------
```

Once the trajectory has been determined, calculating the complex amplitude is very simple:

```
%------------------------------------------------------------------
% calculate and plot scattering amplitude
%------------------------------------------------------------------
% calculate amplitude
A = exp(-1i*(DK(1)*x + DK(2)*y));

% plot real part of amplitude vs time
figure();
plot(t, real(A));
xlabel('t /microseconds'); ylabel('Re(A) /arb. units');
%------------------------------------------------------------------
%------------------------------------------------------------------
```

producing the figure in the margin. There is little information that can be extracted from this graph!

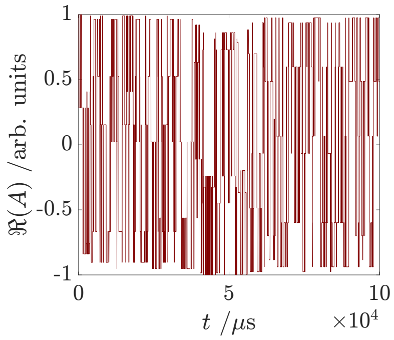

Figure 8 | Plot of the real part of the scattering amplitude against time for a single adsorbate on a square grid, with a parallel momentum transfer of $\Delta K = 2\,[1\ 0]$.

Task 2: Question

Using the expression for the ISF in terms of $A(\Delta\mathbf{K}, t)$, calculate the ISF for that particular value of Δk. Note that the timebase of the ISF is half that of the amplitude. Why is this?

(a) Redefine your ISF to only include the first time-base, then plot the real part of your ISF against time for the new time-base. Your plot should look like an exponential decay, descending into noise with increasing time.

(b) Use the matlab "`fit()`" function to fit

$$I(\Delta\mathbf{K}, t) = e^{-\alpha(\Delta K)t}$$

to the calculated ISF, and plot the fitted curve over the simulation curve to check that it fits theoretical prediction.

Task 2: Solution

We simply continue the above code to calculate the IFT as defined in the theoretical background. Note that we normalise the ISF since the units are arbitrary. This will make fitting the exponential curve easier (only one fitting parameter instead of two). We restrict our attention to half the simulated timebase due to the nature of the Fourier transform.

```
%----------------------------------------------------------------
% ISF calculation
%----------------------------------------------------------------

% evaluate formula
step_1 = fft(A);
step_2 = abs(step_1).^2;
I = ifft(step_2);

% redefine timebases
t = t(1:floor(n_t/2));
I = I(1:floor(n_t/2));

% take normalised real part
ReI = real(I)/real(I(1));

%----------------------------------------------------------------
%----------------------------------------------------------------
```

We then plot the real part of the ISF (you may plot the imaginary part if you wish but it is generally assumed that the experiment is designed to measure the real part unless otherwise stated):

```
%----------------------------------------------------------------
% ISF plot
%----------------------------------------------------------------

figure();
plot(t, ReI); hold on;

%----------------------------------------------------------------
%----------------------------------------------------------------
```

The plot is shown below along with the fitted curve.

The next part is to calculate a fit. Note that an (estimated) initial parameter value is provided so that the fitting function is less likely to diverge and terminate before finding the correct value.

```
%----------------------------------------------------------------
% fit ISF to exponential decay and plot
%----------------------------------------------------------------

% fit exponential decay
g = fittype('exp(-a*x)');
fit_g = fit(t, ReI, g, 'startpoint', [0.001]);

% dephasing rate extracted from fit
alpha = coeffvalues(f);

% plot fitted curve
plot(t, exp(-alpha.*t)); % alternatively, write plot(fit_g)
xlabel('t /microseconds'); ylabel('Re(I) /arb. units');  hold
    off;

%----------------------------------------------------------------
%----------------------------------------------------------------
```

which produces the figure in the margin.

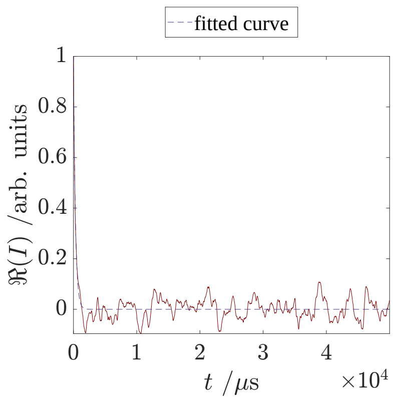

Figure 9 | Plot of the real part of the ISF against (spin-echo) time for a single adsorbate on a square grid, with a parallel momentum transfer of $\Delta K = 2[1\ 0]$.

Task 3: Question

Try varying the following parameters and observe how they affect the ISF:

- Lattice parameter a
- Parallel momentum transfer ΔK
- Hopping probability p
- Scattering direction arg $(\Delta\mathbf{K})$
- Timestep between points in the trajectory
- Total length of the simulation

Task 3: Solution

This is largely left as an exercise to the reader, however to make comparisons between your plots you must ensure you are using the same path each time. The original `mcs` object will then be safe in the workspace for you to use each time you run the file. The persistent workspace *can* be useful every now and then!

Task 4: Question

By iterating over the code you have already written, either with a loop or using vector operations, calculate a fitted value α for a series of values of ΔK, then plot α as a function of ΔK. You should find that your plot has a similar form to the single-jump diffusion model.

Task 4: Solution

This can be achieved simply by putting the existing code in a for loop and restructuring some commands slightly. This is given in a single listing below:

```
% avoids a cluttered workspace
clear;

%----------------------------------------------------------------
% define simulation parameters
%----------------------------------------------------------------

% hop probability
p = 0.01;

% lattice parameter /A
a = 2.5;

% number of timesteps in trajectory simulation
n_t = 100000;

% length of timesteps /micro s
dt = 1;

%----------------------------------------------------------------
```

```
%---------------------------------------------------------------


%---------------------------------------------------------------
% define set of momentum transfers we are interested in
%---------------------------------------------------------------

% number of different moduli DK we wish to find the ISF for
n_DK = 100;

% set of those moduli in given range /A^{-1}
mods_DK = linspace(0, 6, n_DK);

% direction of DK /rad
arg_DK = 0;

% unit vector in given direction
unitvec = [cos(arg_DK) sin(arg_DK)];

% scaled set of vectors with direction arg_DK and moduli mods_DK
DKs = mods_DK' * unitvec;

%---------------------------------------------------------------
%---------------------------------------------------------------


%---------------------------------------------------------------
% iterate over each momentum transfer to calculate a fitted dephasing rate alpha(DK) for each one
%---------------------------------------------------------------

for j=1:n_DK
    DK = DKs(j,:);

    %-----------------------------------------------------------
    % run single MC simulation and calculate scattering amplitude
    %-----------------------------------------------------------

    % simulate
    mcs = MCSimu_sqr(p, a, dt, n_t);
    mcs.nStates_loop(n_t);

    % extract trajectory
    x = mcs.x; y = mcs.y; t = mcs.t;

    % calculate scattering amplitude
    A = exp(-1i*(DK(1)*x + DK(2)*y));

    %-----------------------------------------------------------
    %-----------------------------------------------------------


    %-----------------------------------------------------------
    % ISF calculation
    %-----------------------------------------------------------

    % evaluate ISF formula
    step_1 = fft(A);
    step_2 = abs(step_1).^2;
    I = ifft(step_2);

    % half timebase
    t = t(1:floor(n_t/2)); I = I(1:floor(n_t/2));

    % take normalised real part
    ReI = real(I)/real(I(1));

    %------------------------------------------------------------
    %------------------------------------------------------------



    %------------------------------------------------------------
    % perform exponential fit of ISF
    %------------------------------------------------------------

    % attempt fit
    g = fittype('exp(-a*x)');
    fitg = fit(t, ReI, g, 'startpoint', [0.001]);

    % extract dephasing rate alpha for this momentum transfer DK
    alpha(j) = coeffvalues(fitg);

    %------------------------------------------------------------
    %------------------------------------------------------------
end

%----------------------------------------------------------------
%----------------------------------------------------------------
```

We now have a set of alpha and corresponding moduli of momentum transfers, so we are able to plota dephasing rate curve:

```
%----------------------------------------------------------------
% plot dephasing rate curve
%----------------------------------------------------------------

figure();
plot(mods_DK, alpha); hold on;
xlabel('parallel momentum transfer'); ylabel('dephasing rate');

%----------------------------------------------------------------
%----------------------------------------------------------------


%----------------------------------------------------------------
% fit curve to dingle-jump diffusion model
%----------------------------------------------------------------

% fit single-jump diffusion model
h = fittype('c*(1-cos(b*x))');
fith = fit(mods_DK', alpha', h, 'startpoint', [2*pi/a 0.01]);

% plot fitted curve
plot(fith); hold off;

%----------------------------------------------------------------
%----------------------------------------------------------------
```

Note that the period is $L \approx 2.51 = 2\pi/a$, i.e. the separation between reciprocal lattice sites along the [1 0] azimuth (as described in the theoretical background). Similarly, along the [1 1] azimuth the interplanar

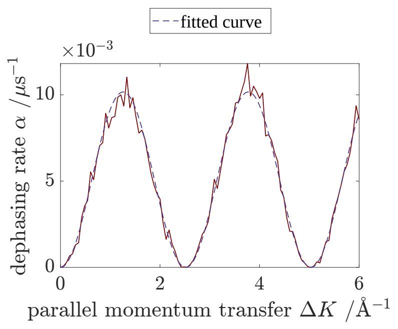

Figure 10 | Plot of dephasing rate against parallel momentum transfer (in the [1 0] direction) for a single adsorbate on a square grid.

spacing of (1 1) planes is $a/\sqrt{2}$, so the period is $L \approx 3.55 = 2\pi/(a/\sqrt{2})$. Play around with different azimuths and verify that this relation holds.

Task 5: Question

Try plotting the same curve for triangular and hexagonal lattices. What do you notice about the new dephasing rate curves?

Task 5: Solution

Along the [1 0] azimuth the curves are as follows: Triangular lattice:

In the triangular lattice there are two jumps of different $x-$lengths, contributing to the appearance of higher Fourier components in this curve for the rate along the [1 0] azimuth

Hexagonal Lattice:

The hexagonal curve has a drastically different curve due to the fact that it is not a Bravais lattice - rather it is a triangular lattice with a motif of two oppositely-oriented sites. Note, however, that the zero point of both curves is the same!

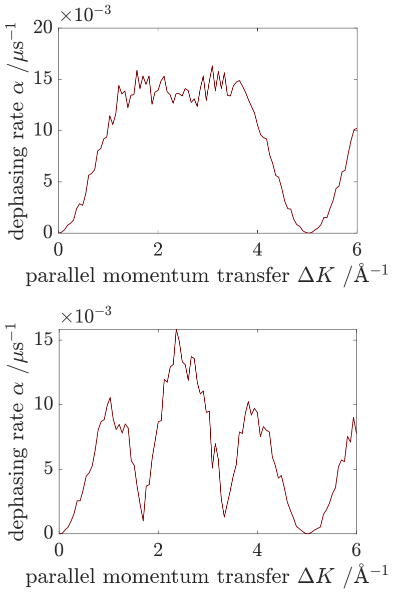

Figure 11 | Plot of dephasing rate against parallel momentum transfer (in the [1 0] direction) for a single adsorbate on a triangular grid (top) and a hexagonal grid (bottom).

Task 6: Question

You may notice that, despite following the theoretical curve on average, the dephasing curves become significantly more noisy for lower numbers of timesteps `n_t`. Why is this?

Task 6: Solution

For a greater number of timesteps, the ISF includes more correlations between different timesteps, so the calculated ISF converges to the analytic formula.