ASSIGNMENT 6: SOLUTION

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

Written in XƎLaTeX

THE CHUDLEY-ELLIOTT MODEL

PREREQUISITES

The previous assignments, and the chapter on *Scattering Functions* from the *Theory Handbook*.

INTRODUCTION

It should not surprise the reader that atoms or molecules may be present on the surface of a solid. To be studied using the helium spin-echo technique, we must obtain the intermediate scattering function (ISF) for their trajectories. This model yields the ISF using the Chudley-Elliot model for surface diffusion - where the diffusion occurs in "jumps" between nearby sites.

THE CHUDLEY-ELLIOT MODEL

The Chudley-Elliott model is an analytical model used to describe the "jump diffusion" of a particle between equivalent adsorption sites in a lattice. What does this statement mean? A set of adsorption sites is considered equivalent if:

- They are energetically identical.
- They form a Bravais lattice with a single site basis (such that each site is a lattice point).

Defining "jump diffusion" is easier. This is the diffusion (*i.e.* "spreading out") of adsorbates on a surface, arising from the instataneous hopping of the adsorbates between nearby sites. The Chudley-Elliott model allows us to generate an analytical expression for the intermediate scattering function (ISF) for a given adsorbate-substrate system. We can compare this function to the ISF data obtained in helium spin-echo spectroscopy measurements, allowing the model to be put under scrutiny in experiment.

Quantitative Description

The Chudley-Elliot model has a simple quantification. We will explore this by first considering the positions that the particles can jump to. Suppose that a particle, initially at the origin, can jump to a finite number of equivalent sites. Let there be n of these equivalent sites. Each site has a position vector $\mathbf{l}_k$, where the subscript index k is an integer between 1 and n (inclusive). What use is the content so far? It can be used to calculate the ISF, allowing us to form an expression relevent to spin-echo spectroscopy. The intermediate scattering function $I(\Delta\mathbf{K}, t)$ is then given by the equation:

$$\frac{\partial}{\partial t} I(\Delta\mathbf{K}, t) = \frac{1}{n\tau} \sum_k I(\Delta\mathbf{K}, t)(e^{-i\Delta\mathbf{K}\cdot\mathbf{l}_k} - 1),$$

where τ is the “site residence time” (a parameter that is inputted into the model), and $\Delta\mathbf{K}$ is the momentum transfer parallel to the plane. The solution to this equation, yielding the ISF, is:

$$I(\Delta\mathbf{K}, t) = I(\Delta\mathbf{K}, 0)\, e^{-\alpha(\Delta\mathbf{K})t},$$

where the dephasing constant $\alpha(\Delta\mathbf{K})$ is given by:

$$\alpha(\Delta\mathbf{K}) = \sum_k 2\Gamma_k \sin^2\left(\frac{\Delta\mathbf{K}\cdot\mathbf{l}_k}{2}\right),$$

where Γ_k is the jump rate to the kth adsorption site.

The dephasing constant α is of interest. It is clear that if we obtain spin-echo data, an exponential function can be fitted with the results. The exponential decay constant can be obtained from this fitting! When calculating the decay constant using the analytic formula, we can split the adsorption sites into shells: each shell consists of a set of sites, all of which are the same distance from the origin. We then assume that the jump rate Γ_k to the kth site, from the origin, is equal for sites in the same shell; we also assume that there is a quantitative relationship between the jump rates for neighbouring shells.

How do we find this relationship between jump rates? We define the probability that the adsorbate jumps from the origin to the ith shell as p_i. We assume that the probabilities of an atom jumping to the 1st, 2nd and 3rd shells from the origin take the form:

$$p_1 = \frac{1}{n};$$
$$p_2 = \frac{s}{n};$$
$$p_3 = \frac{s^2}{n},$$

where the subscript i on p_i represents the ith shell. In these expressions, s is known as the survival probability; this takes a value between 0 and 1. What does this system of equations mean? The most straightforward interpretation of these probabilities is that the probability of jumping to further shells scales as a geometric series. Noting that the common ratio for this series is smaller than 1, it becomes evident that this series represents the exponentially decaying likelihood of jumping as the shell number increases (see appendices).

Can these probabilities be put to use? In addition to assuming the forms of the first three probabilities of jumping, we assume that the probability of the adsorbate jumping any further than the third shell is zero. Consequently, since the total probability of a jump occuring is 1, we see that the sum of the first three probabilities is unity:

$$p_1 + p_2 + p_3 = 1,$$

and so the survival probability for this system satisfies

$$n = 1 + s + s^2.$$

The interpretation of this result is that we can calculate the exact values of each of the probabilities p_1, p_2 and p_3.

We may wish to calculate the jump rate Γ_{ki} for the kth site within the ith shell. It is conceptually straightforward to calculate this quantity. We hope the reader can be pursuaded that the probability of jumping to a particular site is the product of the probability of jumping to the shell the site is in, and the proobability of landing in that site in the shell (rather than any of the others in the shell). This leads to the jump rate Γ_{ki} being given by:

$$\Gamma_{ki} = \Gamma_{tot} p_i \frac{1}{N_i}$$

where the total jump rate Γ_{tot} is:

$$\Gamma_{tot} = \frac{1}{\tau},$$

and N_i is the number of sites in the ith shell.

TASK

Finding the decay constant

1. Using the Chudley-Elliott model, evaluate the dephasing constant $\alpha(\Delta\mathbf{K})$ for a Ru(0001) hexagonal surface. Assume that only the top sites are used as adsorption sites.
   You should consider the momentum transfers $\Delta\mathbf{K}$ that are in the direction of the high-symmetry azimuths, $\langle 11\bar{2}0\rangle$ and $\langle 1\bar{1}00\rangle$, and whose magnitudes are between 0Å^{-1} and 4Å^{-1}.

2. Produce suitable plots of your results.

Tips

You may wish to use some of the functions provided in the solutions to previous tutorials to speed up the coding process. After considering how to approach the task, it is worth looking at the suggested approach below. Consider how your approach compares to the one suggested before starting to code.

We make one more note: a diagram of the shells in a Ru(0001) surface is included in the margin. It is not necessary to reproduce the plot, but it is worth noting the number, and positions, of the sites in each shell. This will allow you to verify your code.

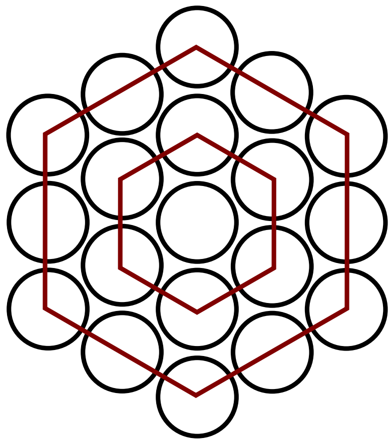
Figure 1 | The first two shells in the close-packed Ru(0001) surface. These are made up of adsorption wells located at the top sites of the surface planes.

Suggested Approach

The approach used in our solution is based around two functions: the so-called `lattice_shell` function, and the `ChudleyElliott` function. Upon writing these functions, your script must call the functions - in the appropriate manner, using the required arguments.

The `lattice_shell` function receives a matrix containing the coordinates of n lattice points (which must include all of the lattice points in the first 3 shells) and returns a $3 \times n$ matrix. The rows of this matrix give, for each lattice point:
1. the x-coordinate,
2. the y-coordinate,
3. the shell number.

Note that these are assigned in this specific order.

Should you opt to use this approach, when writing your function, it is important to note that the function needs to account for how the lattice points in your input matrix may not be ordered by shell number. As described in the theoretical background, we assume that the probability of a jump any further than the third shell is zero. This means that we are only interested in lattice points within the first three shells! It would therefore make sense to order the lattice points by shell number. This will allow it to be easy to see which lattice points are relevant to the model!

The input of the `ChudleyElliott` function is the matrix returned by the `lattice_shell` function, along with a matrix specifying which momentum transfers are to be processed, and a vector of jump rates to each of the first three shells. The Chudley-Elliott function returns the exponential decay constant $\alpha(\Delta\mathbf{K})$. The outputted matrix in our case should be an $m \times 2$ matrix. Each of the two columns in this matrix contains values for the decay constant $\alpha(\Delta\mathbf{K})$, sampled along the two different high-symmetry azimuths at an integer number m values between 0Å^{-1} and 4Å^{-1}.

SUMMARY

This tutorial introduces the Chudley-Elliott model, describing diffusion as jumps between sites on a surface. This provides an analytic expression for the intermediate scattering function, which holds well when compared with experiment.

FURTHER READING: NON-BRAVAIS LATTICES

As discussed in the *Chudley-Elliott* section, the Chudley-Elliott model works well for jump diffusion between equivalent adsorption sites. This makes the model rather restrictive; in most real adsorbate-substrate systems, the adsorbate can attach to more than one type of site. For example, there are many systems where the adsorbate could attach to the two types of hollow site on a close-packed plane.

Modelling such a system requires the consideration of jump diffusion within a non-Bravais lattice. In this case, the Chudley-Elliott model fails. However, there is a way of extending the model to apply to non-Bravais lattices; this approach is discussed in Tuddenham et al (2010).

SOLUTIONS

FINDING THE DECAY CONSTANT: MAIN TASKS

The majority of the work used in this task is generating the functions at the end of the document. Aside from this, all we need to do is generate the inputs into the functions. The first step we need to approach is to find the positions of the lattice points on our surface. We are constructing a lattice from the top sites of a Ru(0001) surface, which is a close packed plane. The positions of these top sites can be found using the `unitCellVecs` and `spanVecs` function listed at the end of this document. The `top_sites` matrix will be one of the inputs into the `lattice_shell` function. Then our positions are found with the code:

```
%-------------------------------------------------------------
% initialise vectors and variables, Finding the positions of the
     top sites
%-------------------------------------------------------------

% Ru{0001} lattice constant
a = 2.55;

% no need to rotate the unit-cell
RotM = 0 ;

% lattice and reciprocal lattice vectors
[a1, a2, b1, b2] = unitCellVecs(1,a,RotM);

% positions of top sites
top_sites = spanVecs(a1,a2,-4:4,-4:4);

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

We then call the `lattice_shell` function, specifying that we are only interested in the first three shells. The `info_atom_shell` will be one of the inputs into the `ChudleyElliott` function:

```
%-------------------------------------------------------------
% Calling lattice_shell function
%-------------------------------------------------------------

% shells we are interested in
shells = [1 2 3];

% generating info_shell matrix
[info_atom_shell] = lattice_shell(top_sites', shells);
```

```

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

Next, we need to create the matrix of parallel momentum transfers `dK`, which will be another input into the `ChudleyElliott` function. This needs to be an $m \times 2 \times 2$ matrix, where m is the number of ΔK values we wish to sample:

```
%------------------------------------------------------------
% Define matrix of parallel momentum transfers dK
%------------------------------------------------------------

% magnitude of dK for both azimuths
dK_mag = 0:0.1:4;

% unit vector in <1 1 -2 0> direction
b10_unit = b1./norm(b1);

% unit vector in <1 -1 0 0 > direction
b11_unit = (b1+b2)./norm(b1+b2);

% values of dK along < 1 1 -2 0> direction
dK(:,:,1) = b11_unit.*dK_mag';

% values of dK along < 1 -1 0 0 > direction
dK(:,:,2) = b10_unit.*dK_mag';

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

We then need to define the jump rates, using the formulae given in the section on the theoretical background. Note that both the site residence time `tau` and the survival probability `s` are paramaters that can be changed. Then the jump rates are calculated with:

```
%------------------------------------------------------------
% Define jump probabilities (for 1st, 2nd, 3rd shells)
%------------------------------------------------------------

% survival probability (can be varied)
s = 0.4;

%defining n (the reciprocal of the first term in the geometric
    series for the probabilities)
n = 1 + s + s^2;

% [p1 p2 p3] are jump probabilities to shells [ 1, 2, 3 ]
% p_jump is the vector [p1 p2 p3]
p_jump = [1/n s/n s^2/n];

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




%------------------------------------------------------------
% Define jump rates
%------------------------------------------------------------
```

```
% N_shell(i) is the number of sites in the ith shell
N_shell = zeros(size(shells));

% iterate through each shell, calculate the number of sites in
    the ith shell and amend the vector with this info
for i=1:length(shells)
    N_shell(i) = sum(info_atom_shell(3,:)==shells(i));
end

% site residence time in [ps]
tau = 4;

% the total jump rate is gamma_total, with units of [ps^{-1}]
gamma_total = 1/tau;

% gamma_i(i) is the jump rate to the ith shell
gamma_i = gamma_total*p_jump./N_shell;

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

We now have everything we need to call the `ChudleyElliott` function and plot out our results. Thus achieving both tasks in one go, we list the code:

```
%-----------------------------------------------------------------
% call chudley elliott function
%-----------------------------------------------------------------

% Generate alpha(dK), the array of values of the dephasing rate
    alpha (in GHz)
alpha_GHz = ChudleyElliott(info_atom_shell, dK, shells, gamma_i)
    ;

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




%-----------------------------------------------------------------
% Plot
%-----------------------------------------------------------------

% initialise a new figure
fig_CE = figure();

% initialise first subplot
sp_CE(1) = subplot(1,2,1);

% plot the dephasing rate for < 1 1 -2 0 > as a function of
    parallel momentum transfer
plot(dK_mag, alpha_GHz(:,2));

% put a title on the first subplot
title('$\langle 11\bar{2}0 \rangle$',"Interpreter","latex");

% initialise the second subplot
sp_CE(2) = subplot(1,2,2);

% plot the dephasing rate for < 1 -1 0 0 > as a function of
```

```
    parallel momentum transfer
plot(dK_mag,alpha_GHz(:,1));

% put a title on the second subplot
title('$\langle 1\bar{1}00 \rangle$',"Interpreter","latex");

% Making the plot prettier
YLim_max = max([sp_CE(1).YLim(2) sp_CE(2).YLim(2)]);

% Set the limits on the first subplot
sp_CE(1).YLim = [0 YLim_max];

% Set the limits on the second subplot
sp_CE(2).YLim = [0 YLim_max];

% Set the position of the Y axis on the second subplot
sp_CE(2).YAxisLocation = "right";

% Set the positions of the two subplots
sp_CE(1).Position = [0.1 0.1 0.4 0.8];
sp_CE(2).Position = [0.5 0.1 0.4 0.8];


% reverse the orientation of the x axis on the first plot
sp_CE(1).XDir = "reverse";


% iterate through the subplots, and label the axes on each
    subplot
for i=1:2
    xlabel(sp_CE(i),'$\Delta \mathrm{K}$',"Interpreter","latex")
    ylabel(sp_CE(i),'ISF Decay Rate $\alpha$ / GHz',"Interpreter
        ","latex")

% end for loop
end

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

Functions Used

We will show you how to write the two key functions: the `lattice_cell` function, and the `ChudleyElliott` function. Then we will show you how to write the main script.

First, we will consider how to write the `lattice_shell` function. As always when writing a function, it is important to start your code with a comment outlining the purpose of the function, and the form that its inputs and outputs take

```
%-----------------------------------------------------------------
% define the lattice_shell function
%-----------------------------------------------------------------

function [info_atom_shell] = lattice_shell(lattice, shell_num)
% lattice_shell(lattice, shell_num)
% This script returns the lattice information for the required
    shell number(s)
```

```
%
% Inputs:
% - lattice : 2xn matrix with x-y coordinates of the n lattice
    points
% - shell_num : 1xn matrix of the required n nearest neighboring
     atoms.
%
% Output:
%  - info_atom_shell : a 3xn matrix containing:
%    1st row: atoms in x axis
%    2nd row: atoms in y axis%
%    3rd row: the shell the atom is in
%
```

Our first task is to set up the output matrix, and assign values to rows 1 and 2. This is a simply a case of transferring information from the (inputted) `lattice` matrix to the output matrix:

```
    %Setting up info_atom_shell

    %Creating empty 3xn matrix
    info_atom_shell = zeros( 7, size( lattice, 2 ) );

    %Setting values for rows 1 and 2
    info_atom_shell( 1 : 2, : ) = lattice;
```

Next, we need to think about how to organise these lattice points by shell number. A shell is defined as a collection of lattice points that are the same distance from the origin, so it makes sense to create variables that contain information about the distance of each lattice point from the origin:

```
    %Setting up variables that will help us organise lattice
        points by shell

    %Distance of each lattice point from origin
    r = sqrt(lattice( 1, : ).^2 + lattice( 2, : ).^2);

    %A list of the unique values of r
    [rUnique]=uniquetol(r, 1e-3);

    %Removing the rUnique value corresponding to lattice point
        at origin
    rUnique(1) = [];

    %Variable contains distance between origin and 1st shell
    shell_boundaries = rUnique(1);
```

We now need to organise the lattice points by shell number. We do this by assuming that a lattice point in shell number i should be a distance of `shell_boundaries * i` from the origin. On the `i`th cycle of the for loop, we check whether each lattice point is in shell `i` and encode the result in a variable called `indx`, which assigns a value of 1 to a lattice point that is in shell `i` and a value of 0 to a lattice point that isn't. This is achieved with:

```
    %Organising the lattice points by shell number

    %This for loop finds which lattice points are in each shell and fills in row 3 accordingly
    for i = 1:length(shell_num)

        % obtain the shell number
        n_shell = shell_num(i);

        % Lattice points {in}/{not in} shell i assigned value of {1}/{0}
        indx = r > shell_boundaries*(n_shell-1)+1e-3 & r <= shell_boundaries*n_shell+1e-3;

        % notify if shell is empty
        if isempty(find(indx)), error('Shell is empty.');

        % end the if statement
           end

    % The lattice points in shell i are assigned their shell number i
    info_atom_shell(3,indx) = n_shell;

    % end the for loop
    end
```

We then remove those lattice points that haven't been assigned (because they are in shells that we aren't interested in) and sort the remaining lattice points by shell number:

```
%Assigning a value of 1 to lattice points that haven't been assigned to a shell
indx = info_atom_shell(3,:) == 0;

%Removing unassigned lattice points
info_atom_shell(:,indx) = [];

%Sorting by shell number
[~,indx] = sort(info_atom_shell(3,:));

%Reordering info_atom_shell by shell number
info_atom_shell = info_atom_shell(:,indx);


% end lattice shell function
end

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

Next, we need to consider how to write the `ChudleyElliott` function. The Chudley-Elliott formula used to evaluate $\alpha(\Delta\mathbf{K})$ is essentially a sum over each of the allowed jump vectors. Therefore, we need a for loop over each allowed jump vector, and within each cycle of the for loop we need to calculate a single term in the sum for all values of ΔK. All of this needs to be nested within a for loop over all of the directions along which we want to evaluate $\alpha(\Delta\mathbf{K})$:

```
%-----------------------------------------------------------------
% define the chudley elliott function
```

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

function alpha_GHz = ChudleyElliott(info_atom_shell,dk,
    shells_requested,ni_jump)
% CHUDLEYELLIOT computes the alpha(dk) for jump on a bravais
    lattice to
% allowed shells, for given jump probabilities.
%
% Input parameters:
% - info_atom_shell : 3xN matrix, where n is the numb of lattice
     points in a given lattice.
% - info_atom_shell(:,k)={x,y,shell number} for the kth lattice
    point
% - dK(i,j,k) : momentum transfer, {Kx,Ky,azimuth index}
% - shells_requested: a 1xn matrix of the shells to which jumps
    are allowed
% - ni_jump : A vector of jump rates from (0,0) to all the
    resquested shells.
% -  ni_jump(i) is the jump rate to a lattice point in the ith
    shell
%
% Output:
% - alpha_Ghz : value of alpha for each value of dK in Ghz

% Initialise a matrix for the dephasing rate alpa(dk) of size
    axb, where b is number of azimuths to sample and a is number
     of dk values for each azimuth
alpha = zeros( size(dk, 1), size(dk, 3) );

% For each azimuth, calculate the entry in the dephasing rate
    matrix (i.e. in alpha)
for i=1:size(dk, 3)

    %Creating an ax2 matrix containing the dkx and dky values to
         sample along a particular azimuth
    dK_ = dk( :, :, i );

    % For all the allowed jumps
    for j=1:size( info_atom_shell,2 )

        % Find the shell of jump 'j' and store its number in
            shell_n
        shll_n = info_atom_shell(3,j);

        %If the shell number is 0 then continue (skip to next
            iteration of for loop)
        if shll_n == 0
            continue;

        % otherwise end the if statement
        end

        % If the shell number isn't one of those requested, then
             continue
        ni_jump_indx = find(shll_n == shells_requested,1);

        % check if it is requested - if it is, then it is found
            using the function above, otherwise False
        if isempty(ni_jump_indx)
            continue;
        % otherwise end condition
        end

        % if the shell number is requested, then set ni as the jumps
        ni = ni_jump(ni_jump_indx);

        % Evaluating Chudley Elliot formula
        product_dK_J = dK_(:, 1)*info_atom_shell(1, j) + dK_(:, 2)*info_atom_shell(2, j);


        alpha_shell = 2*ni*(sin(product_dK_J/2)).^2;

        % add to the dephasing rate matrix outside of this for loop, iterate to next instance i between 1 and size(dk, 3)
        alpha(:,i) = alpha(:,i) + alpha_shell;

    % end the for loop
    end

% end the for loop
end

%Converting to Ghz
alpha_GHz = alpha * 1e3;


% end the chudley elliott function
end

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

Once these two functions have been defined, the main script simply has to generate the necessary inputs for these functions. This is achieved with the code above.

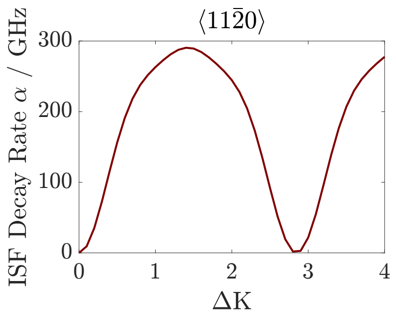

Figure 2 | A plot of the dephasing rate α against the parallel momentum transfer ΔK in the $\langle 1120\rangle$ direction. The site residence time is 4ps and the surivival probability is 0.4

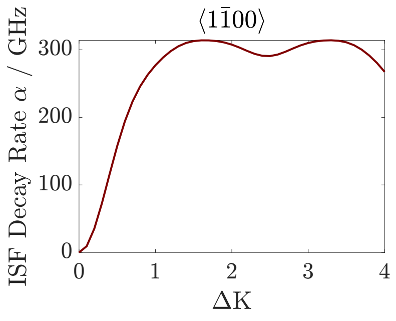

Figure 3 | A plot of the dephasing rate α against the parallel momentum transfer ΔK in the $\langle 1100\rangle$ direction. The site residence time is 4ps and the surivival probability is 0.4

APPENDICES

The listing for the omnipresent `auxCls` class is given below.

```
%-----------------------------------------------------------------
% The auxCls class listing
% Contains:
% - unitCellVecs
% - real2recip
% - sampleUnitCell
% - spanVecs
%-----------------------------------------------------------------
classdef auxCls
    properties(Constant)
    % no special properties to define
    end

    % define the functions, one by one
    methods(Static)

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

```
% first function: unitCellVecs
%---------------------------------------------------------
function [a1, a2, b1, b2] = unitCellVecs(surfaceType, lttcCnst, rotation)
    % UNITCELLVECS creates the vectors of a unit cell (Real & Reciprocal space)
    % Currently, two types of symmetries are supported, hexagonal and fcc. Also. tt is possible to define rotation of the unit cell.
    % Input:   surfaceType - 1-hexagonal 2-fcc
    %          lttcCnst - Lattice constant
    %          rotation - either a 2x2 rotational Matrix, or the rotation in degrees.
    % Output: {a1,a2}  - real space vectors
    %         {b1,b2}  - reciprocal vectors

    % This section determines what the surface type is. This is used to structure the real space vectors accordingly

    % Substrate vector
    if length(surfaceType) == 3
        angle=surfaceType(3);
        Rmat = [cosd(angle) sind(angle); -sind(angle) cosd(angle)];
        a1s = [1 0]*surfaceType(1);
        a2s = [Rmat*[1 0]']'*surfaceType(2);

    % if surface type is 1, then the surface is hexagonal. Basis vectors are (orienting one along x): x + 0y and -0.5x + root3 / 2 y
    elseif surfaceType == 1

        % first vector: x + 0y
        a1s=[1 0]*lttcCnst;

        %second vector: -0.5x + root3 / 2 y
        a2s=[-0.5 sqrt(3)/2]*lttcCnst;

    % if surface type is 2, then the surface is fcc(100). This is a square lattice! so vectors are x + 0y and 0x + y
    elseif surfaceType == 2

        % first vector is x + 0y, magnitude lttcCnst
        a1s=[1 0]*lttcCnst;

        % second vector is 0x + y, magnitude lttcCnst
        a2s=[0 1]*lttcCnst;

    % end set of if statements
    end



    % This section rotates the unit cell vectors

    % assume rotation angle in degrees
    if length(rotation)==1,
        Rmat = [cosd(rotation) sind(rotation); -sind(rotation) cosd(rotation)];
```

```
            a1 = [Rmat*a1s']';
            a2 = [Rmat*a2s']';
        else
            % Adsorbate vectors
            a1 = rotation(1,1)*a1s + rotation(1,2)*a2s;
            a2 = rotation(2,1)*a1s + rotation(2,2)*a2s;
        end


        %find reciprocal vectors
        b1=2*pi*cross([a2 0],[0 0 1])/([a1 0]*[cross([a2
            0],[0 0 1])]');
        b1=b1(1:2);


        b2=2*pi*cross([0 0 1],[a1 0])/([a2 0]*[cross([0 0
            1],[a1 0])]');
        b2=b2(1:2);
    end
    %--------------------------------------------------------
    %--------------------------------------------------------





    %--------------------------------------------------------
    % real2recip function
    %--------------------------------------------------------
    % finds the reciprocal vectors
    function [d1,d2]=real2recip(c1, c2, toRecip)
        if toRecip
            d2=2*pi*cross([0 0 1],[c1 0])/norm(cross([c1
                0],[c2 0]));
            d2=d2(1:2);
            d1=2*pi*cross([c2 0],[0 0 1])/norm(cross([c1
                0],[c2 0]));
            d1=d1(1:2);
        else
            disp('real2recip: not implemented yet ...')
        end
    end
    %--------------------------------------------------------
    %--------------------------------------------------------



    %--------------------------------------------------------
    % sampleUnitCell function
    %--------------------------------------------------------
    function  R = sampleUnitCell(len, a1, a2)
        % Used to be called uniSpacedPointsInUnitCell
        % Uniformly distributed over the unit cell
        vec=0:1/len:1-1/len;
        len=length(vec);
        [N,M]=meshgrid(vec);
        for n=1:len
            for m=1:len
                R((n-1)*len+m,1:2)=N(n,m)*a1+M(n,m)*a2;
            end
        end
    end
    %-------------------------------------------------------
    %-------------------------------------------------------


    %-------------------------------------------------------
    % spanVecs function
    %-------------------------------------------------------
    function G = spanVecs(b1,b2,nVec1,nVec2)

        [N_g,M_g]=meshgrid(nVec2,nVec1);
        Gx = N_g * b1(1) + M_g * b2(1);
        Gy = N_g * b1(2) + M_g * b2(2);
        G = [Gx(:) Gy(:)];
    end
    %-------------------------------------------------------
    %-------------------------------------------------------

% end of functions
end

end
%---------------------------------------------------------------
%---------------------------------------------------------------
```