ABSTRACT

The random walk problem finds applications in diverse fields, *it is central to statistical physics*. It is *essential* in predicting how fast one gas will diffuse into another, how fast heat will spread in a solid, how big fluctuations in pressure will be in a small container, and many other statistical phenomena. Here we investigate the problem of shielding a nuclear reactor by applying the random walk problem to the neutrons being emitted from the reactor core. We use a 2-dimensional array to represent the shielding and study the characteristics of the system as different parameters such as thickness of shielding, kinetic energy of neutrons, density of the shielding material etc. are varied. We also investigate the trajectory of the neutrons that leave the reactor and enter the shielding.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1. Random walk Problem

A man starts from a point 0 and walks x yards in a straight line; he then turns through any angle whatever and walks another x yards in a straight line. He repeats this process n times. This is the gist of a random walk problem. Various types of random walks are of interest, which can differ in several ways. A popular random walk model is that of a random walk on a regular lattice, where at each step the location jumps to another site according to some probability distribution. In a **simple random walk**, the location can only jump to neighboring sites of the lattice, forming a lattice path. This is what we will be using throughout this project.

2. Physics with Random-walk

In physics, random walks are used as simplified models of physical Brownian motion and diffusion such as the random movement of molecules in liquids and gases. See for example diffusion-limited aggregation. Also, in physics, random walks and some of the self-interacting walks play a role in quantum field theory. Random walks, in polymer physics, describes an ideal chain. It is the simplest model to study polymers. Furthermore, random walks underlie the method of Fermi estimation. More practical applications of percolation theory are discussed below.

3. Random walk Problem Applications

An elementary example of a random walk is the random walk on the integer number line, , which starts at 0 and at each step moves +1 or −1 with equal probability. Other examples include the path traced by a molecule as it travels in a liquid or a gas (see Brownian motion), the search path of a foraging animal, the price of a fluctuating stock and the financial status of a gambler: all can be approximated by random walk models, even though they may not be truly random in reality.

As illustrated by those examples, random walks have applications to engineering and many scientific fields including ecology, psychology, computer science, physics, chemistry, biology, economics, and sociology. Random walks explain the observed behaviours of many processes in these fields, and thus serve as a fundamental model for the recorded stochastic activity.

1. Brownian motion

Real gas molecules can move in all directions, not just to neighbours on a chessboard. We would therefore like to be able to describe a motion similar to the random walk above, but where the molecule can move in all directions. A realistic description of this is Brownian motion - it is similar to the random walk (and in fact, can be made to become equal to it. See the fact box below.), but is more realistic. In the beginning of the twentieth century, many physicists and mathematicians worked on trying to define and make sense of Brownian motion - even Einstein was interested in it!

To get started, the following is a simulation of a gas, and one particle is marked in yellow. Its path describes a Brownian motion B_t at time t.

How can we define Brownian motion? Let's think about the movement of the gas molecule during a small time-interval from time t1 to time t2. We measure its position at times t1 and t2, but not in between. Between times t1 and t2, the molecule will have bumped into other particles randomly and gotten kicks in random directions. The longer this time interval is, the farther will the molecule have travelled between our measurements. It would therefore make sense that $B_{t_2} - B_{t_1}$ should somehow be a random quantity that increases as $t_2 - t_1$ does. Choosing the right random quantity is what defines a Brownian motion: we define, $B_{t_2} - B_{t_1} = N(0, t_2 - t_1)$ where $N(0, t_2 - t_1)$ is a normal distribution with $t_2 - t_1$ variance

Now, Einstein realized that even though the movements of all the individual gas molecules are random, there are some quantities we can measure that are not random, they are predictable and can be calculated. One such quantity is the density ρ of the gas molecules. Einstein showed that the density satisfies a differential equation

$$\frac{\partial \rho}{\partial t} = D\frac{\partial^2 \rho}{\partial x^2},$$

called the diffusion equation, and where D is the diffusion coefficient that can be calculated. This is an equation that can be solved, so we are able to predict something with certainty from a random model - this is an example of the strategy that is used in statistical mechanics. Einstein's equation showed that diffusion processes, for instance seeing a drop of ink spread out in water, are caused by Brownian motion.

2. Image segmentation

In image segmentation, random walks are used to determine the labels (i.e., "object" or "background") to associate with each pixel. This algorithm is typically referred to as the random walker segmentation algorithm. In the first description of the algorithm, a user interactively labels a small number of pixels with known labels (called seeds), e.g., "object" and "background". The unlabeled pixels are each imagined to release a random walker, and the probability is computed that each pixel's random walker first arrives at a seed bearing each label, i.e., if a user places K seeds, each with a different label, then it is necessary to compute, for each pixel, the probability that a random walker leaving the pixel will first arrive at each seed. These probabilities may be determined analytically by solving a system of linear equations. After computing these probabilities for each pixel, the pixel is assigned to the label for which it is most likely to send a random walker. The image is modeled as a graph, in which each pixel corresponds to a node which is connected to neighboring pixels by edges, and the edges are weighted to reflect the similarity between the pixels. Therefore, the random walk occurs on the weighted graph

3. Random walk theory in Economics

The Random Walk Theory, or the Random Walk Hypothesis, is a mathematical model of the stock market. Proponents of the theory believe that the prices of securities in the stock market evolve according to a random walk.

A “random walk” as we know is a statistical phenomenon where a variable follows no discernible trend and moves seemingly at random. The random walk theory, as applied to trading, most clearly laid out by Burton Malkiel, an economics professor at Princeton University, posits that the price of securities moves randomly, and that, therefore, any attempt to predict future price movement, either through fundamental or technical analysis, is futile.

Since the Random Walk Theory posits that it is impossible to predict the movement of stock prices, it is also impossible for a stock market investor to outperform or “beat” the market in the long run. This implies that it is impossible for an investor to outperform the market without taking on large amounts of additional risk. As such, the best strategy available to an investor is to invest in the market portfolio, i.e., a portfolio that bears a resemblance to the total stock market and whose price reflects perfectly the movement of the prices of every security in the market.

4. Other Applications

- In brain research, random walks and reinforced random walks are used to model cascades of neuron firing in the brain.
- In vision science, ocular drift tends to behave like a random walk. According to some authors, fixational eye movements in general are also well described by a random walk.
- In psychology, random walks explain accurately the relation between the time needed to make a decision and the probability that a certain decision will be made.
- Random walks have also been used to sample massive online graphs such as social networking services
- On the web, the Twitter website uses random walks to make suggestions of whom to follow

4. Experiments with Numerical Simulation

A numerical simulation is a calculation that is run on a computer following a program that implements a mathematical model for a physical system. Numerical simulations are required to study the behavior of systems whose mathematical models are too complex to provide analytical solutions, as in most nonlinear systems.

Computer simulation developed hand-in-hand with the rapid growth of the computer, following its first large-scale deployment during the Manhattan Project in World War II to model the process of nuclear detonation. It was a simulation of 12 hard spheres using a Monte Carlo algorithm. Computer simulation is often used as an adjunct to, or substitute for, modeling systems for which simple closed form analytic solutions are not possible. There are many types of computer simulations; their common feature is the attempt to generate a sample of representative scenarios for a model in which a complete enumeration of all possible states of the model would be prohibitive or impossible.

Computer simulations are used in a wide variety of practical contexts, such as:

- analysis of air pollutant dispersion using atmospheric dispersion modelling
- design of complex systems such as aircraft and also logistics systems.
- design of noise barriers to effect roadway noise mitigation
- modelling of application performance[15]
- flight simulators to train pilots
- weather forecasting
- forecasting of risk
- simulation of electrical circuits
- Power system simulation

4.1 Numerical simulation approaches for random-walk problem

There are a wide variety of algorithms and lattices random walk can be applied to, however, as said earlier, random walk on a regular lattice, where at each step the location jumps to another site according to some probability distribution is what we are going to be dealing with. It is also important for the algorithm to be as efficient as possible because we are dealing with lattices of large dimensions and a large number of particles.

Here, we will be implementing our random walk on a 2-dimensional square lattice, hence, traversing a 2-D matrix will suffice

Traversing a 2-dimensional matrix

The fundamental operation involved in a random walk is traversing the array, here we are dealing with a 2-dimensional square lattice, the below algorithm represents 2-d array traversal.

Here A is a two – dimensional array with M rows and N columns. This algorithm traverses array A and applies the operation PROCESS to each element of the array.

```
1. Repeat For I = 1 to M
2. Repeat For J = 1 to N
3. Apply PROCESS to A[I][J]
   [End of Step 2 For Loop]
   [End of Step 1 For Loop]
4. Exit
```

4. Random Number

Randomness is important while modeling real world phenomenon, in the case of random-walk problem, the direction of traversal needs to be randomly decided at each step, hence, there is a requirement for generation of a random number to decide the future direction each time we occupy a new cell in the lattice.

In MATLAB we have the *rand* function which produces a random real value between 0.00 and 1.00. Usage of this is elaborated in section 2.2.

5. Programming with MATLAB

MATLAB is a proprietary multi-paradigm programming language and numeric computing environment developed by MathWorks. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages.

The overview below will discuss some of the essential syntax of MATLAB used in this project, MATLAB was chosen due to its ease of use and superior features.

```
! This is a comment. who % Displays all variables in memory
whos % Displays all variables in memory, with their types
clear % Erases all your variables from memory
clear('A') % Erases a particular variable
openvar('A') % Open variable in variable editor

clc % Erases the writing on your Command Window
diary % Toggle writing Command Window text to file
ctrl-c % Abort current computation

edit('myfunction.m') % Open function/script in editor
type('myfunction.m') % Print the source of function/script to Command Window

profile on  % turns on the code profiler
profile off     % turns off the code profiler
profile viewer  % Open profiler

help command    % Displays documentation for command in Command Window
doc command     % Displays documentation for command in Help Window
lookfor command % Searches for command in the first commented line of all functions
lookfor command -all % searches for command in all functions


% Output formatting
format short    % 4 decimals in a floating number
format long     % 15 decimals
format bank     % only two digits after decimal point - for financial calculations
fprintf('text') % print "text" to the screen
disp('text')    % print "text" to the screen

% Variables & Expressions
myVariable = 4  % Notice Workspace pane shows newly created variable
myVariable = 4; % Semi colon suppresses output to the Command Window
4 + 6       % ans = 10
8 * myVariable  % ans = 32
2 ^ 3       % ans = 8
a = 2; b = 3;
c = exp(a)*sin(pi/2) % c = 7.3891

% Calling functions can be done in either of two ways:
% Standard function syntax:
load('myFile.mat', 'y') % arguments within parentheses, separated by commas
% Command syntax:
load myFile.mat y   % no parentheses, and spaces instead of commas
% Note the lack of quote marks in command form: inputs are always passed as
% literal text - cannot pass variable values. Also, can't receive output:
[V,D] = eig(A);  % this has no equivalent in command form
[~,D] = eig(A);  % if you only want D and not V



% Logicals
1 > 5 % ans = 0
10 >= 10 % ans = 1
3 ~= 4 % Not equal to -> ans = 1
3 == 3 % equal to -> ans = 1
3 > 1 && 4 > 1 % AND -> ans = 1
3 > 1 || 4 > 1 % OR -> ans = 1
~1 % NOT -> ans = 0
```

```
% Logicals can be applied to matrices:
A > 5
% for each element, if condition is true, that element is 1 in returned matrix
A( A > 5 )
% returns a vector containing the elements in A for which condition is true

% Strings
a = 'MyString'
length(a) % ans = 8
a(2) % ans = y
[a,a] % ans = MyStringMyString


% Cells
a = {'one', 'two', 'three'}
a(1) % ans = 'one' - returns a cell
char(a(1)) % ans = one - returns a string

% Structures
A.b = {'one','two'};
A.c = [1 2];
A.d.e = false;

% Vectors
x = [4 32 53 7 1]
x(2) % ans = 32, indices in Matlab start 1, not 0
x(2:3) % ans = 32 53
x(2:end) % ans = 32 53 7 1

x = [4; 32; 53; 7; 1] % Column vector

x = [1:10] % x = 1 2 3 4 5 6 7 8 9 10
x = [1:2:10] % Increment by 2, i.e. x = 1 3 5 7 9

% Matrices
A = [1 2 3; 4 5 6; 7 8 9]
% Rows are separated by a semicolon; elements are separated with space or comma
% A =

%     1     2     3
%     4     5     6
%     7     8     9

A(2,3) % ans = 6, A(row, column)
A(6) % ans = 8
% (implicitly concatenates columns into vector, then indexes into that)


A(2,3) = 42 % Update row 2 col 3 with 42
% A =

%     1     2     3
%     4     5     42
%     7     8     9

A(2:3,2:3) % Creates a new matrix from the old one
%ans =

%     5     42
%     8     9

A(:,1) % All rows in column 1
%ans =

%     1
%     4
%     7

A(1,:) % All columns in row 1
%ans =

%     1     2     3    m = func3(3,2,k)
    function func2(a,b,c) result(f)    !return variable declared to be 'f'.
        implicit none
        integer, intent(in) :: a,b
```

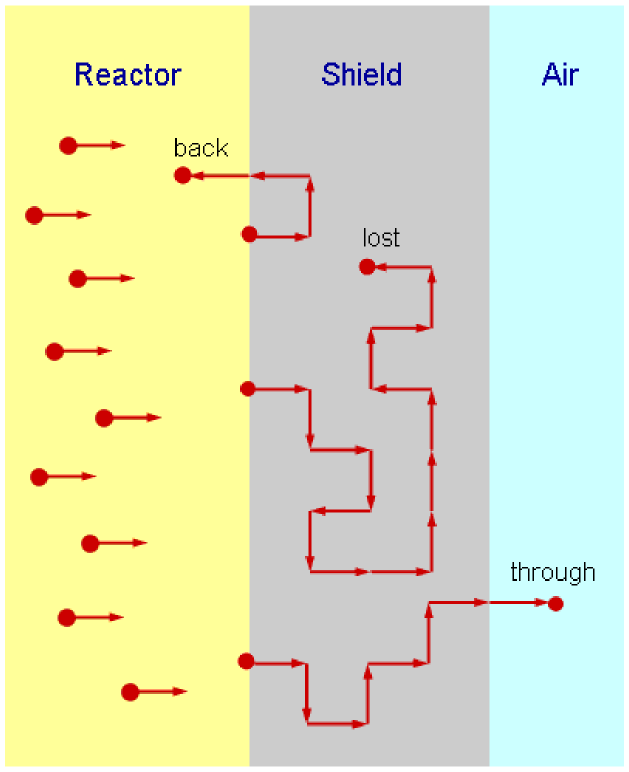
```
 function output = double_input(x)
    %double_input(x) returns twice the value of x
    output = 2*x;
end
double_input(6) % ans = 12

% You can also have subfunctions and nested functions.
% Subfunctions are in the same file as the primary function, and can only be
% called by functions in the file. Nested functions are defined within another
% functions, and have access to both its workspace and their own workspace.
% If you want to create a function without creating a new file you can use an
% anonymous function. Useful when quickly defining a function to pass to
% another function (eg. plot with fplot, evaluate an indefinite integral
% with quad, find roots with fzero, or find minimum with fminsearch).
% Example that returns the square of its input, assigned to the handle sqr:
sqr = @(x) x.^2;
sqr(10) % ans = 100
doc function_handle % find out more
% User input
a = input('Enter the value: ')
% Stops execution of file and gives control to the keyboard: user can examine
% or change variables. Type 'return' to continue execution, or 'dbquit' to exit
keyboard
% Reading in data (also xlsread/importdata/imread for excel/CSV/image files)
fopen(filename)
% Output
disp(a) % Print out the value of variable a
disp('Hello World') % Print out a string
fprintf % Print to Command Window with more control
% Conditional statements (the parentheses are optional, but good style)
if (a > 23)
    disp('Greater than 23')
elseif (a == 23)
    disp('a is 23')
else
    disp('neither condition met')
end
% Looping
% NB. looping over elements of a vector/matrix is slow!
% Where possible, use functions that act on whole vector/matrix at once
for k = 1:5
    disp(k)
end

k = 0;
while (k < 5)
    k = k + 1;
end


% Common matrix functions
zeros(m,n) % m x n matrix of 0's
ones(m,n) % m x n matrix of 1's
diag(A) % Extracts the diagonal elements of a matrix A
diag(x) % Construct a matrix with diagonal elements listed in x, and zeroes elsewhere
eye(m,n) % Identity matrix
linspace(x1, x2, n) % Return n equally spaced points, with min x1 and max x2
inv(A) % Inverse of matrix A
det(A) % Determinant of A
eig(A) % Eigenvalues and eigenvectors of A
trace(A) % Trace of matrix - equivalent to sum(diag(A))
isempty(A) % Tests if array is empty
all(A) % Tests if all elements are nonzero or true
any(A) % Tests if any elements are nonzero or true
isequal(A, B) % Tests equality of two arrays
numel(A) % Number of elements in matrix
triu(x) % Returns the upper triangular part of x
tril(x) % Returns the lower triangular part of x
cross(A,B) %  Returns the cross product of the vectors A and B
dot(A,B) % Returns scalar product of two vectors (must have the same length)
transpose(A) % Returns the transpose of A
fliplr(A) % Flip matrix left to right
flipud(A) % Flip matrix up to down
```

CHAPTER-2

METHODS

1. The Problem Statement and Model

A beam of neutrons bombards a reactor's wall. Considering motion of neutrons as a random walk on (x,y) plane find probabilities for neutrons (as a function of the shield size etc.):
a) to be back in the reactor
b) to be captured in the shield
c) to get through the shield.



Conditions:
1. Only four directions of motion are possible (left, right, up or down)
2. On the next step the neutron cannot step back, but only forward, left or right
3. The probability to go forward is twice more than changing a direction
4. On each step the neutron loses one unit of energy
5. Initial neutron energy is enough for 100 steps
6. Initial neutron velocity is perpendicular to the shield

2.Setup

To work with MATLAB, very little overhead is necessary, the MATLAB environment on its own is sufficient.

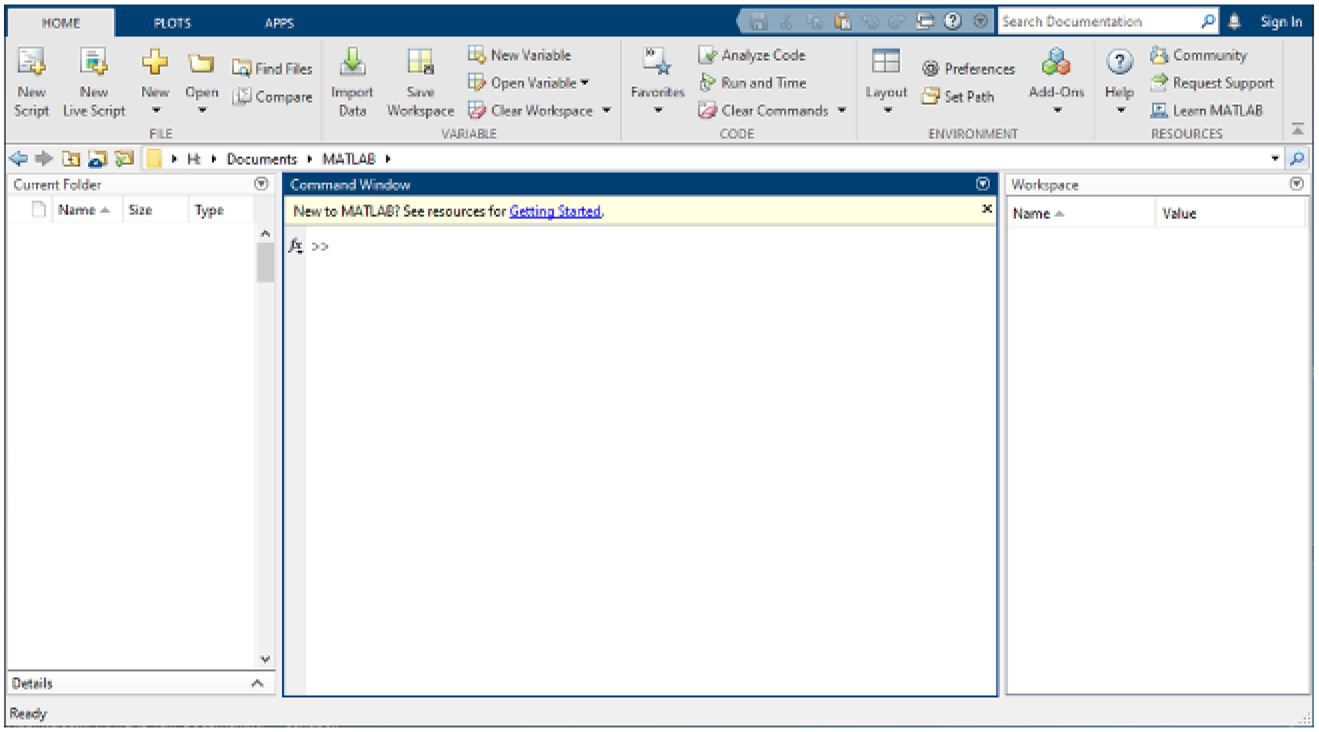
2.1 Installing MATLAB in Windows and running programs.

It is to be noted that MATLAB being proprietary software requires a license to install and use. Installing MATLAB on a Windows machine is a straight forward process

Download the MATLAB® Runtime from the website at *https://www.mathworks.com/products/compiler/matlab-runtime.html*.
Unzip the MATLAB Runtime installer. To unzip the installer:

- Right click the zip file MATLAB_Runtime_R2020b_win64.zip
- Select **Extract All**, and then Double-click the file setup.exe from the extracted files to start the installer.
- When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click Next to proceed with the installation.
- Specify the folder in which you want to install the MATLAB Runtime in the Folder Selection dialog box.
- Confirm your choices and click Next. The MATLAB Runtime installer starts copying files into the installation folder.
- Click Finish to exit the installer.

Go to installation directory and run matlab.exe to start the MATLAB environment.



New to MATLAB? See resources for Getting Started.

```
>>
```

Figure 2.1: MATLAB environment

2.2 Hello World in MATLAB.

A hello world example in MATLAB.
All programming languages are introduced by the most famous computer program on the planet, "Hello world", so let's begin with that for the time being. Open MATLAB on your computer. You will be greeted by a desktop environment of its own. In the center you can see a window labelled as "Command Window". The cursor should be already flickering there; all you now have to do is to write this...

```
disp('Hello world !');
```

and press the return key. The output will be shown just below the command as -

Hello world !

2.3 Random Number generation with MATLAB

In MATLAB we have the ***rand*** function that returns a random number between 0 and 1. **X** = rand returns a single uniformly distributed random number in the interval (0,1). Using this function, we can weight the random value to randomly decide the direction of travel of the neutron according to a set probability for going straight or changing direction to either left or right. It is assumed that the probability of the neutron going straight without deviation is higher than the probability of it going left or right at each step in the matrix. So, let's say probability of neutron going straight is 0.5 and hence probability of it going either left or right will be 0.25 each. The below code snippet shows how this logic is implemented by weighting a random number as discussed above.

2.3.1 MATLAB function to return future direction of travel according to probability of deflection

The below function in *cdrin.m* contains a function that returns the direction of travel of the neutron of the next step it has to traverse through the matrix. It takes the probability of deflection as argument.

```
function[dir]= cdirn(pr_defl) %takes probability of deflection as argument and returns direction of travel
dir=0;
ps=1-pr_defl; %probability of going straight without deflection
pl=pr_defl/2; %probability of going left
pr=pr_defl/2; %probability of going right
rand;
if(rand>=0 && rand<= ps)
dir = 0;
end
if(rand>ps && rand<= pl+ps)
dir=-1;
end
if(rand>pl+ps && rand<=1)
dir=1;
end
end
```

This code decides the future direction of travel of the neutron at each step it makes through the matrix, and according to this direction the next step is taken, this process is repeated until any of our final conditions are met.

2.3.2 MATLAB function to check the position of the neutron in the matrix

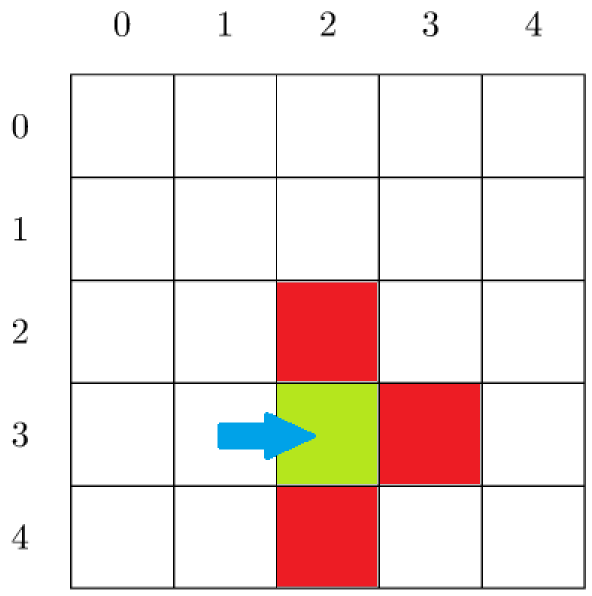
As the neutron takes each step in the matrix, we need to check the location of the neutron to see if it has met any of our end conditions i.e., has the neutron reached the last row? Has it run out of kinetic energy (number of steps), has it gone out of bounds (side hits) or has it returned back to the first row (back to reactor)? We check this at each step by calling a function and then incrementing the counts for each case if a condition has been met and then ending the loop for the current neutron and proceed to the next one starting from the first row.

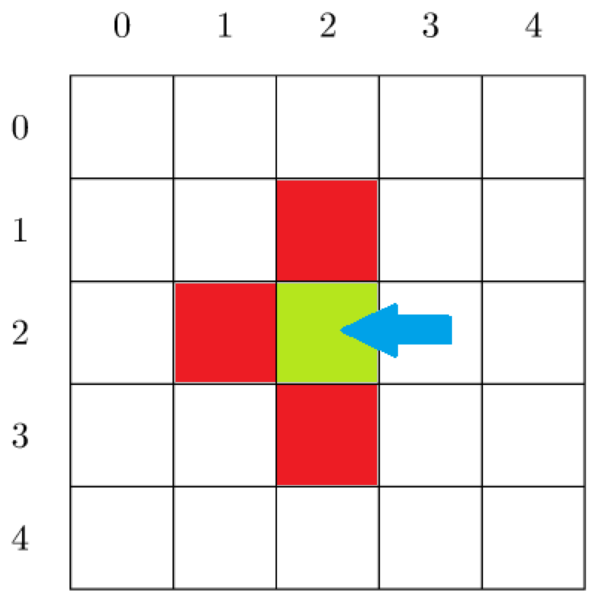
```
function[kill,side_hits,return_count,penetration_count]= curr_status(i,j,m,n,k,k_e)
kill=0;
side_hits=0;
return_count=0;
penetration_count=0;

  if(j<1||j>n && k<=k_e)
  side_hits=1;
  kill=1; %kill is the flag used to end the loop for the current neutron when end condition is met
 end

 if(i<1 && k<=k_e)
 return_count=1; %count is returned and incremented in iter8.m
 kill=1;
 end

 if(i>=m && k<=k_e)
 penetration_count=1;
 kill=1;
 end
```

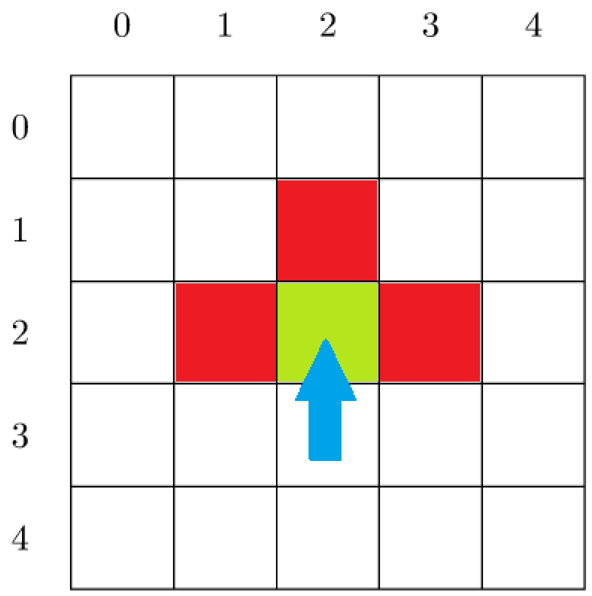
The above function written in *curr_status.m* takes the current coordinates of the neutron [i, j] , the size of the matrix and the kinetic energy of the neutron as argument and checks if any of the end conditions are satisfied for that neutron, if not, the variable kill will have the default 0 value and the neutron can proceed to take the next step, but if an end condition is met, the variable kill will have value 1 and this will be returned to iter8.m along with the respective count which will then be added to the total tally which will thus be incremented by 1.

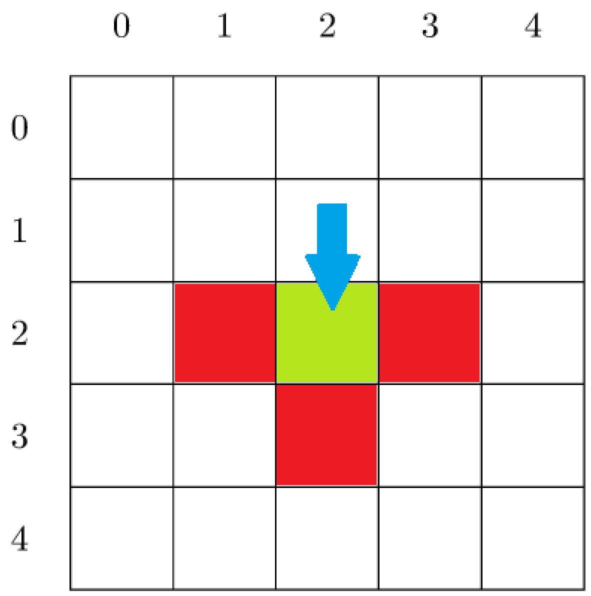
2.3.3 Random walk in a 2-D matrix

There arise 9 cases of traversal depending upon the previous direction of travel and the next direction to proceed in (left, right or straight). The below image depicts all the 9 cases and all 4 directions of travel, here the blue arrow indicates previous direction of travel, green square indicates current position and red squares indicate possible future locations. This logic is incorporated into *main.m* which is given in the following section.









2.3.4 Random walk program in MATLAB.

The below code in *main.m* traverses the 2-D matrix in a random walk according to the deflection probabilities. It accounts for all the 9 possible direction changes discussed in the above section. This program simulates the case for a single neutron, it can be iterated n times to obtain the results for n number of neutrons, we will see this in the upcoming section.

```
function[s_h,r_c,p_c,ke_over]= main(p,k_e,pr_defl)
s_h=0;
%cdirnprev=zeros(10);
r_c=0;
return_count=0;
side_hits=0;
penetration_count=0;
ke_over=0;
p_c=0;
m=p;
n=p;
particle_count=6;
shield=zeros(m,n);
k=2;
w=0;

%while(particle_count>0)
cdirnprev=zeros(1,30);
cdirnprev(1,1)=0;
%clear cdirnprev;
%particle_count=particle_count-1;
j=randi([1,n],1);
w=0;
i=1;
while(k<=k_e)    %KE=k

[w,side_hits,return_count,penetration_count]=curr_status(i,j,m,n,k,k_e);
w;
r_c=r_c+return_count;
p_c=p_c+penetration_count;
s_h=s_h+side_hits;
if(w==1)
break;
end

[dir]=cdirn(pr_defl);
cdirnprev(1,k)=dir;

if(cdirnprev(1,k-1)==0 && dir==0)
i=i+1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end

if(cdirnprev(1,k-1)==0 && dir==1)
j=j+1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end
```

```

if(cdirnprev(1,k-1)==0&&dir==-1)
j=j-1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end



if(cdirnprev(1,k-1)==1&&dir==0)
j=j+1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end

if(cdirnprev(1,k-1)==1&&dir==1)
i=i+1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end

if(cdirnprev(1,k-1)==1&&dir==-1)
i=i-1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end


if(cdirnprev(1,k-1)==-1&&dir==0)
j=j-1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end

if(cdirnprev(1,k-1)==-1&&dir==1)
i=i+1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end

if(cdirnprev(1,k-1)==-1&&dir==-1)
i=i-1;
if(i>=1&&j>=1)
shield(i,j)=particle_count;
end
end
k=k+1;
if(k==k_e&&side_hits+return_count+penetration_count==0)
ke_over=ke_over+1;
end


end
```

Now all that remains to do is to iterate the above code as per the number of neutrons and to store all the variables in a text file for further analysis.

2.4 Program to iterate the random walk for N neutrons

The below program in *iter8.m* runs our random walk simulation for N number of neutrons, it also lets us to set all the input variables and outputs all of the results to a specified text file.

```
%Particle stats keeping number of particles, array size and KE constant
clear all
conf_perc=0;
f=1;
array_size=20;
kin_E=5;
deflec=0.25;
np=10000; %Number of particles
data = zeros(8, np);
    s_h_c=0; %side hit count (should this be considered as 'trapped' ?)
    r_c_c=0; %return to core count
    p_c_c=0; %penetrate count
    ke_over_c=0; %trapped inside count

    fileId = fopen('data_ext.txt', 'w');
    fprintf(fileId, '%s\n', 'f array_size kin_E deflec np s_h_c r_c_c p_c_c ke_over_c conf_perc'); %printing header


    while(f<=np)
    [s_h,r_c,p_c,ke_over]=main(array_size,kin_E,deflec); %array_size, KE,pr_defl
    s_h_c=s_h+s_h_c;
    r_c_c=r_c+r_c_c;
    p_c_c=p_c+p_c_c;
    ke_over_c=ke_over_c+ke_over;
    data(1, f) = f;
    data(2, f) = array_size;
    data(3, f) = kin_E;
    data(4, f) = deflec;
    data(5, f) = np;
    data(6, f) = s_h_c;
    data(7, f) = r_c_c;
    data(8, f) = p_c_c;
    data(9,f) = ke_over_c;
    conf_perc=100*(f/(s_h_c+r_c_c+p_c_c+ke_over_c)); %for debugging...(this must equal n_p...error decreases with higher n_p)
    data(10,f) = conf_perc; %confidence %
    f=f+1;
    end
    fprintf(fileId, '%d %d %d %f %d %d %d %d %d %f\n', data);
    fclose(fileId);
```

USAGE

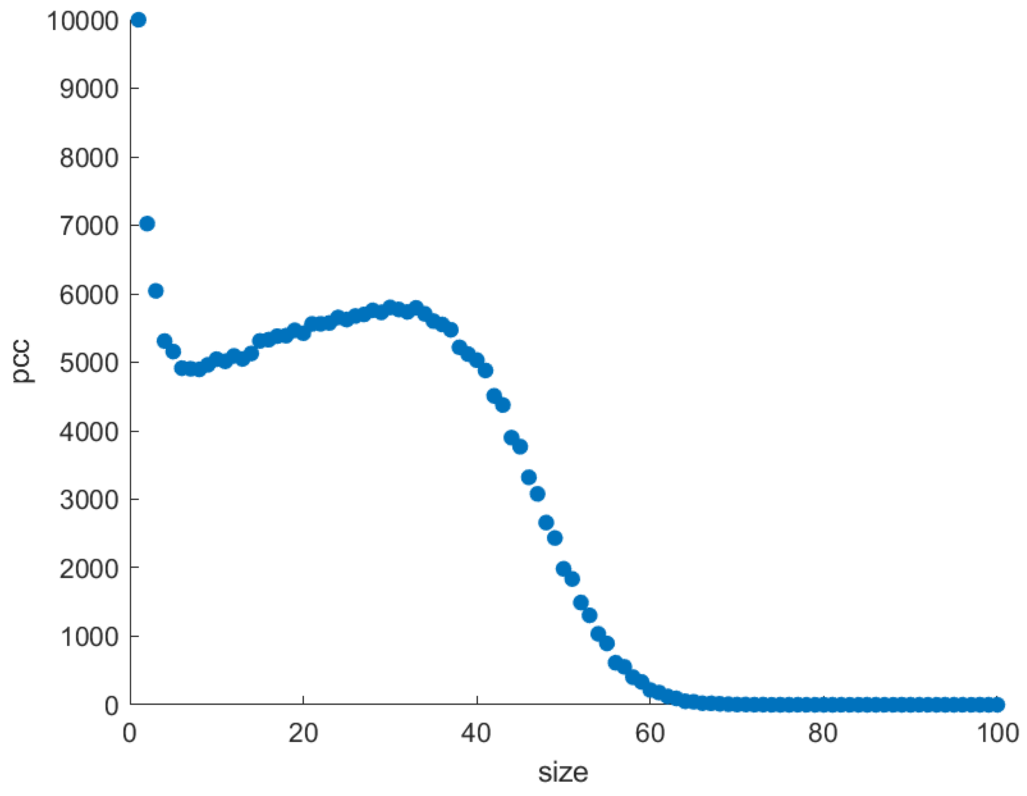
1. *INPUTS***:** *Dimensions of the array, kinetic energy, number of neutrons, deflection probability.*
2. *OUTPUTS: Total number of neutrons that– lost kinetic energy, returned to the reactor and penetrated the shielding.*
3. *The percentage error in the simulation can be found as 100* **-** *conf_perc.*
4. *Results are stored in 'data_ext.txt'.*

CHAPTER-3

RESULTS AND DISCUSSIONS

1. Penetration count vs Size of shielding

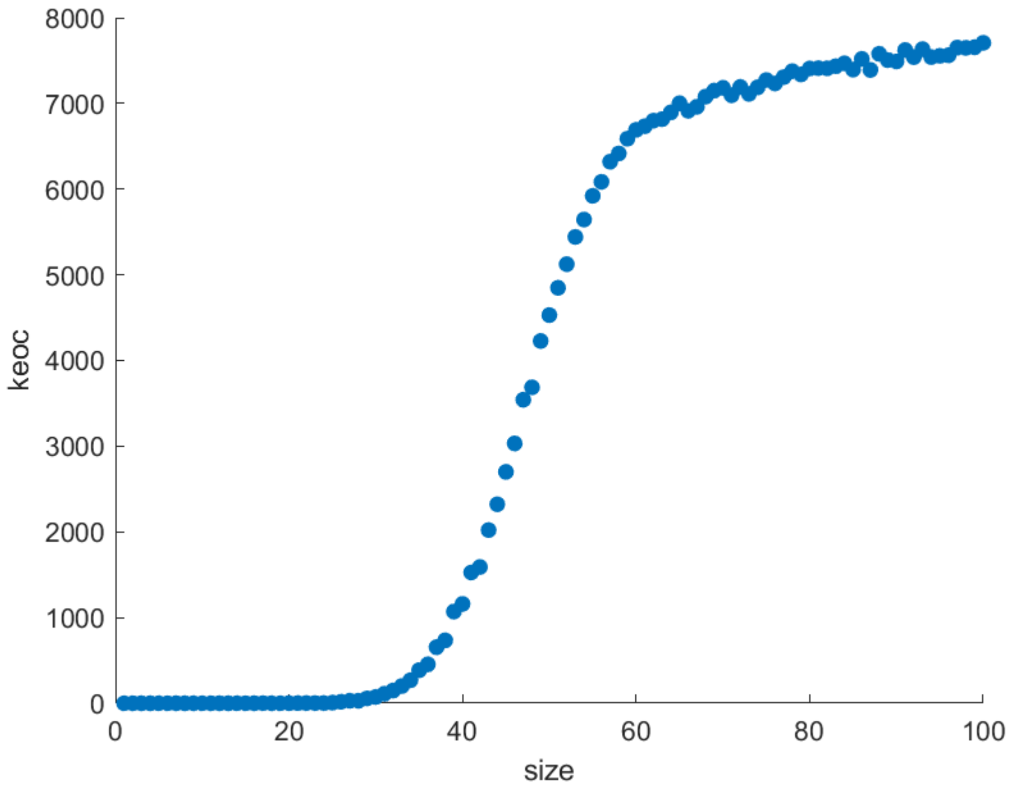
The variation of number of particles that penetrated the shielding was studied as the size of the array was increased for a total of 10,000 particles at kinetic energy of 100 units.



The penetration count was seen to first decrease, then reach a constant value and then decrease to zero as the size of the shielding was increased.

<u>2.Caught in shield count vs Size of shielding</u>

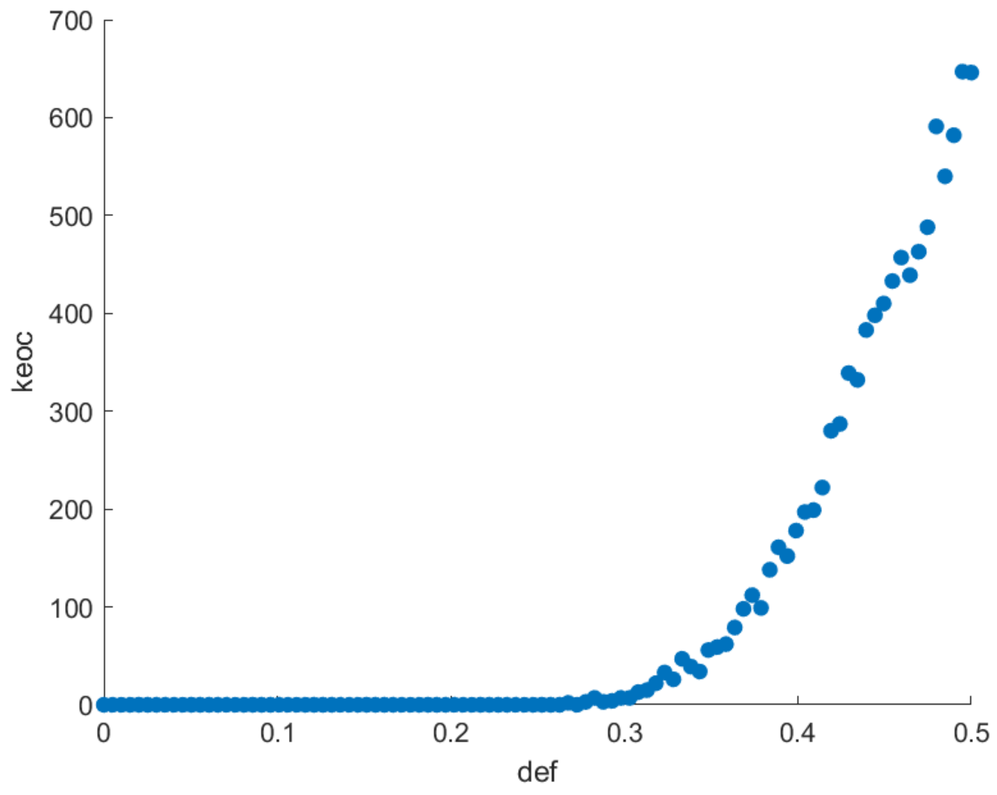
The variation of number of particles that were caught in the shielding was studied as the size of the shielding was increased for a total of 10,000 particles at kinetic energy of 100 units.



The number of particles that get trapped in the shielding was found to increase sharply and reach a saturation value as shown in the plot.

<u>3.Caught in shield count vs Probability of deflection</u>

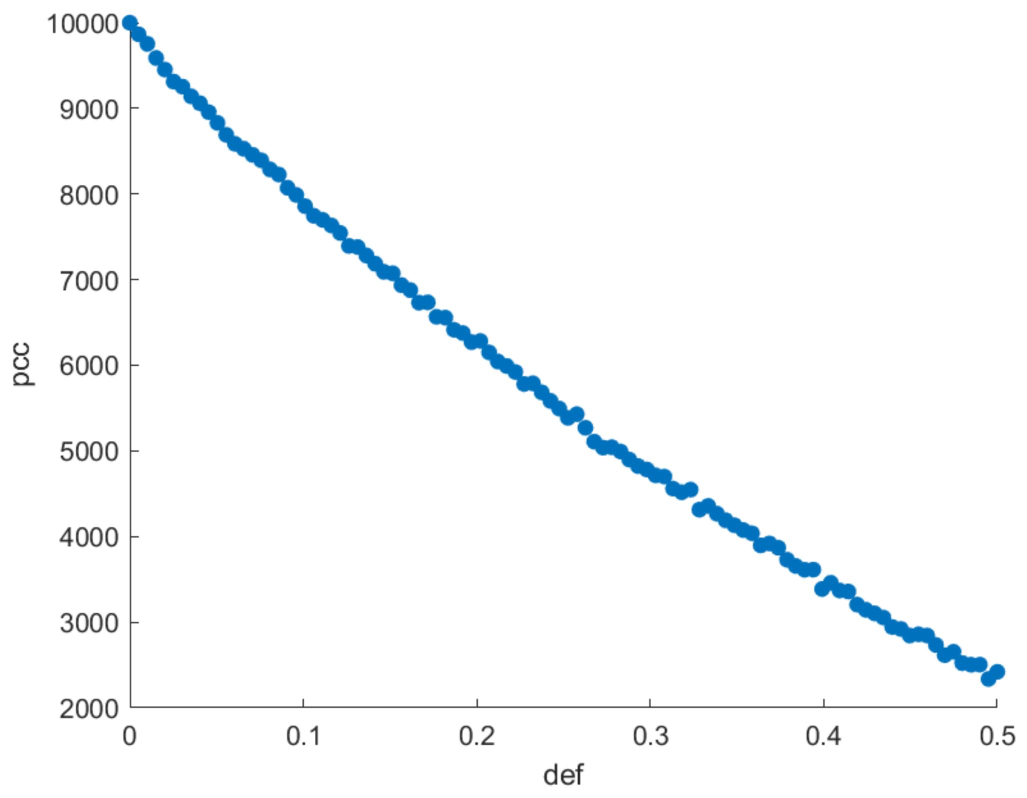
The variation of number of particles that were caught in the shielding was studied as the deflection probability of the neutrons was increased for a total of 10,000 particles at array size 20 and kinetic energy of 100 units.



The number of particles that get trapped increase sharply after the probability of deflection crosses about 30%. This can be inferred as the effect of increasing the density of the shielding material.

4.Penetration count vs Probability of deflection

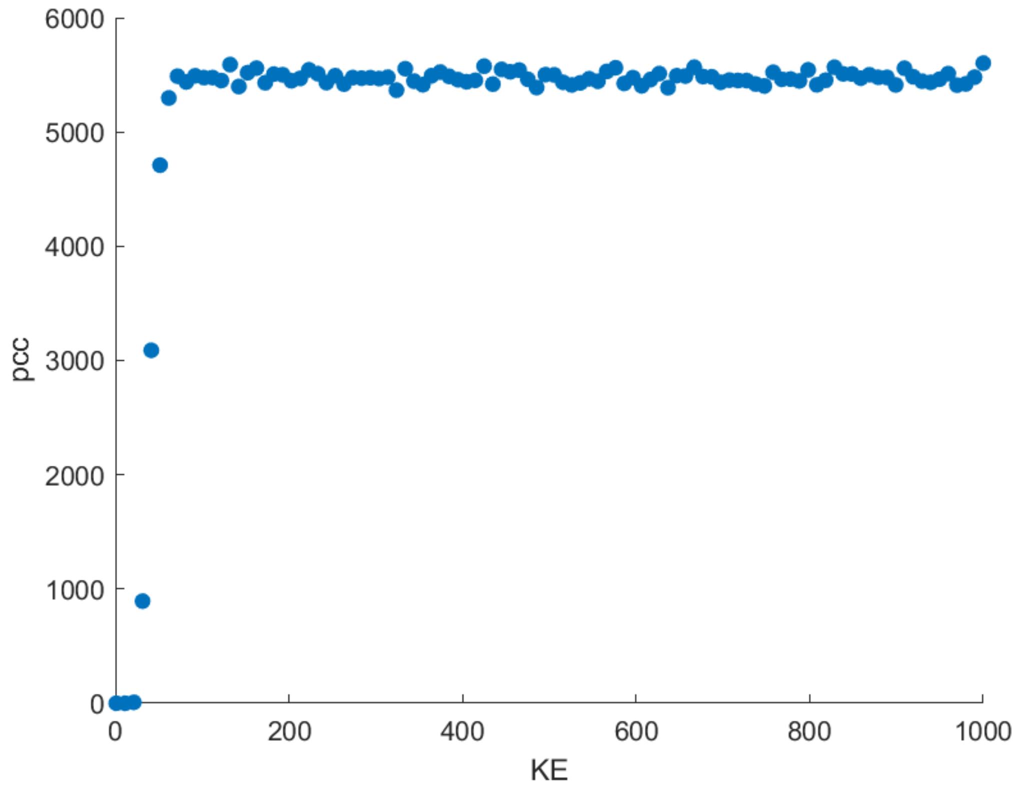
The variation of number of particles that penetrated the shielding was studied as the deflection probability of the neutrons was increased for a total of 10,000 particles at array size 20 and kinetic energy of 100 units.



The penetration count decreased linearly with increasing density of the shielding (deflection probability).

<u>5.Penetration count vs Kinetic energy of neutrons</u>

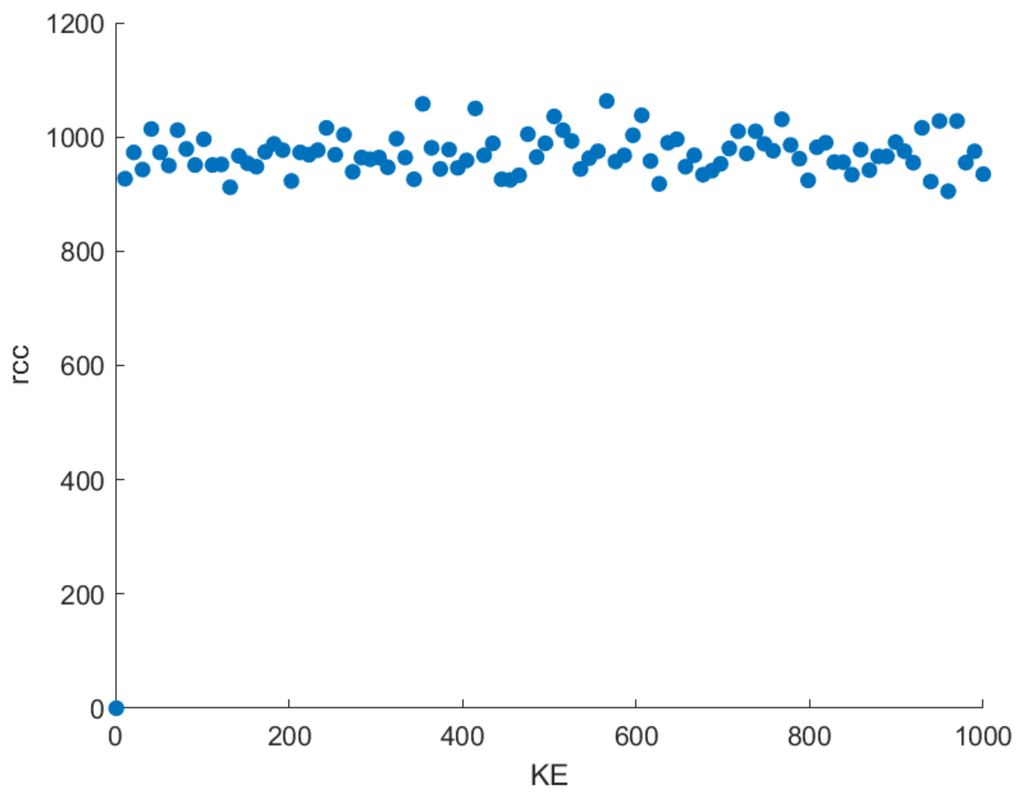
The variation of number of particles that penetrated the shielding was studied as the kinetic energy of the neutrons was increased for a total of 10,000 particles at array size 20.



A sharp increase in penetration count was observed as expected when the kinetic energy increased initially and then it reached a saturation value for kinetic energies greater than 100 units.

6.Return to core count vs Kinetic energy of neutrons

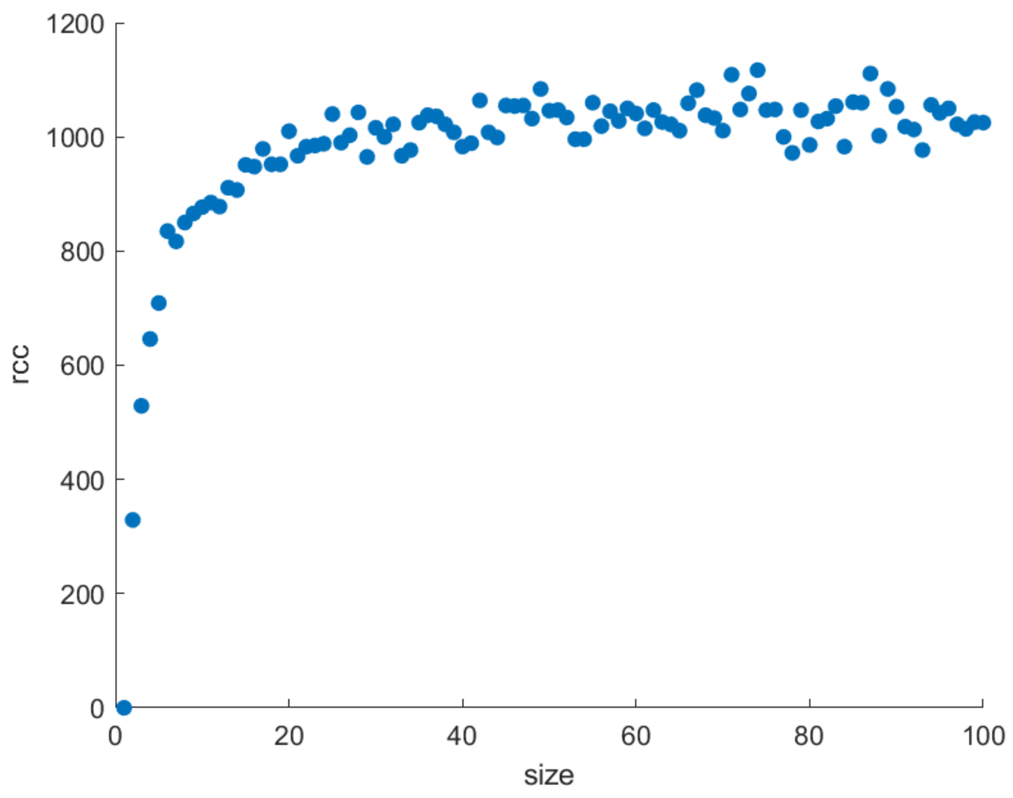
The variation of number of particles that return back to the reactor core was studied as the kinetic energy of the neutrons was increased for a total of 10,000 particles at array size 20.



The return to core count is seen to remain almost constant throughout the range of kinetic energies of the neutrons.

<u>7.Return to core count vs Size of shielding</u>

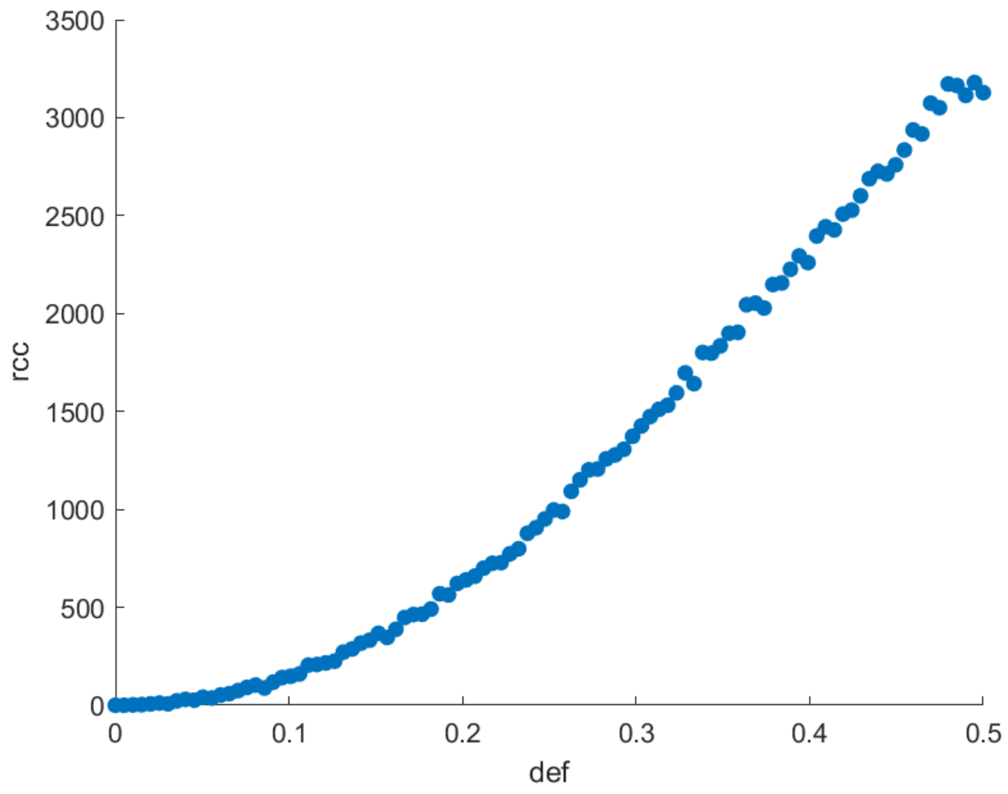
The variation of number of particles that return back to the reactor core was studied as the size of the shielding was increased. Kinetic energy was kept constant at 100 units for a total of 10,000 particles.



Here we can see the return to core count rise sharply and reach a saturation point as we increase the array size.

8.Return to core count vs Probability of deflection

The variation of number of particles that return back to the reactor core was studied as the probability of deflection was increased. Kinetic energy was kept constant at 100 units for a total of 10,000 particles.



Here we can see the return to core count rise as we increase the probability of deflection, here the probability of deflection can be taken as the density of the material, and the results are as expected because as density of material increases, the chances of deflection of the neutron inside the shielding also increases.

CONCLUSION

This report has discussed the study of shielding a nuclear reactor simulated using a 2-D array and a random-walk algorithm. Using this model, some comparisons were done by varying parameters involved - such as size of the matrix, kinetic energy of neutrons, deflection probability etc. and some of the conclusions drawn from the results obtained are discussed below.

(i) The return to core count is seen to increase exponentially and reach a saturation point as we increase the thickness of the shielding (7). It is however seen to increase linearly with increasing density of the material(8). Return to core count does not show appreciable variation with change in Kinetic energy of the neutrons hence we can say it is independent of kinetic energy of the neutrons(6).

(ii) The penetration count is seen to increase sharply with increasing kinetic energy and then reach a saturation value and remains constant(5). The penetration count is also seen to decrease linearly with increasing density of material as can be expected(4). The penetration count as the size of the shielding is varied gives us some interesting results as seen in the first graph, it first decreases sharply, reaches a saturation value and then keeps decreasing back to zero as array size increases(1).

(iii) The number of neutrons that were captured inside the shielding remains constant until a threshold value for size of shielding, but then increases sharply and then reaches a saturation value(2). The number of neutrons that get caught in the shield is also seen to increase as the density of the material is increased(3).

REFERENCE

http://physics.gu.se/~frtbm/joomla/media/mydocs/LennartSjogren/kap2.pdf
https://en.wikipedia.org/wiki/Random_walk#Applications
http://pages.di.unipi.it/ricci/SlidesRandomWalk.pdf
https://www.jstor.org/stable/27851819?seq=1

https://www.researchgate.net/publication/337501163_Random_Walks_A_Review_of_Algorithms_and_Applications

https://en.wikipedia.org/wiki/Random_walk_hypothesis

https://en.wikipedia.org/wiki/Brownian_motion

http://web.mit.edu/8.334/www/grades/projects/projects17/OscarMickelin/brownian.html

https://learnxinyminutes.com/docs/matlab/

https://corporatefinanceinstitute.com/resources/knowledge/trading-investing/what-is-the-random-walk-theory/

https://latexbase.com/d/95038cb5-8fe2-4c9c-94e7-7d6369586331

http://galileo.phys.virginia.edu/classes/152.mf1i.spring02/RandomWalk.htm

https://ww2.odu.edu/~agodunov/teaching/phys420_09/2009_Projects.pdf