ID6105 **Divyanshu Pundir**
Computational Tools **ED15013**

Assignment 2

Pseudo Random Number Generation

Linear congruential generator is used to generate random integers. It follows the scheme:

$$X_{n+1} = (aX_n + c)\ mod\ m$$

where a and c are large integers, X_n are the generated random numbers and X_0 is called the seed.

custom_random.py contains the functions that are used for operations involving random numbers. *randint()* generates and returns an array random numbers in range from low to high (low is included but high is not).

choice() takes an array as input and returns randomly selected element from it.

Code

custom_random.py

```
import numpy as np
import time


# Generates random numbers form [low, high)
# Uses linear congruential generator algorithm
def randint(low, high, size):
    a = 2147483629
    c = 2147483587
    m = 2**31 - 1
    seed = int(10 ** 10 * time.process_time())

    rand = np.zeros(size, dtype=int)
    rand[0] = (a * seed + c) % m

    for i in range(1, size):
        rand[i] = (a * rand[i-1] + c) % m

    # Bringing random numbers to the required range
    rand = low + rand % (high-low)

    return rand


# Choose a random number from an array
def choice(arr):
    arr_len = len(arr)
    rand = randint(0, arr_len, 1)
    return arr[rand[0]]
```

1 Question 1

Question 1 requires us to simulate the different cases of the Monte Hall problem. There are 3 different cases involved in this problem which are discussed below. The problem is simulated using 10,000 iterations. The doors are represented as a set of 3 numbers, $S = \{0, 1, 2\}$.

An array of 10,000 correct answers is created by randomly selecting one element from set S. Similarly, another array of initial selections is created. The number of times the contestant wins is counted and the probability of winning is calculated by dividing it by the total number of simulations for each case.

1.1 Never Swap

In this case the contestant never avails the option of swapping his/her selection. This case is easy to simulate, since removing a wrong answer from the set of possible selections does not change the initial selection. It is a redundant step. Hence, a simple check if the initial selection was the correct one is enough to verify if the selection is correct.

- correct[] = randint(0, 3, num_sim)
- selection[] = randint(0, 3, num_sim)
- wins = 0
- for i in 0 to num_sim - 1
- S = {1, 2, 3}
- remove wrong answer from S which is not selection[i]
- if (selection[i]= correct[i])
- wins ← wins + 1
- return wins / number of simulations

1.2 Always Swap

In this case, the contestant always swaps his/her selection after a wrong option is removed. On observing, it can be seen the contestant will lose only if his/her initial selection is the correct answer. Now, we have to create set of answers from which the host can remove one wrong answer. It cannot contain the correct answer as well as the initial selection. It can be done by removing the correct answer and selection from the set of possible answers. If the length of the array is 2, it means the contestant has selected the correct answer, therefore he/she will lose. If it is 1, then a wrong answer was selected. This means the contestant will win.

- correct[] = randint(0, 3, num_sim)
- selection[] = randint(0, 3, num_sim)
- wins = 0
- for i in 0 to num_sim - 1
- S = {1, 2, 3}
- remove correct[i] and selection[i] from S
- if (length(S) = 1)
- wins ← wins + 1
- return wins / number of simulations

1.3 Coin Toss

In this case, after a wrong answer is removed, the contestant tosses a coin to select between the two remaining doors.

- correct[] = randint(0, 3, num_sim)
- selection[] = randint(0, 3, num_sim)
- wins = 0
- for i in 0 to num_sim - 1
- S = {1, 2, 3}
- remove wrong answer from S which is not selection[i]
- new_selection = choice(S) # Randomly select one from the remaining 2 elements
- if (new_selection = correct[i])
- wins ← wins + 1
- return wins / number of simulations

1.4 Results

The program was run 5 times and the results are as follows:

	Never Swap	Always Swap	Coin Toss
	0.3263	0.6737	0.5042
	0.3229	0.6771	0.5058
	0.3372	0.6628	0.5062
	0.3333	0.6667	0.5067
	0.3331	0.6669	0.5065
Mean =	0.33056	0.66944	0.50588

These results agree with the theoretical values.

1.5 Code

monte_hall.py

```
import numpy as np
import custom_random


def never_swap(correct, selection):
    n = len(correct)
    # Omitting redundant step of removing wrong answer
    num_wins = np.sum(correct == selection)
    return num_wins / n


def always_swap(correct, selection):
    n = len(correct)
    num_wins = 0

    for i in range(n):
        available_choices = [0, 1, 2]
```

```
        available_choices.remove(correct[i])

        if available_choices.__contains__(selection[i]):
            available_choices.remove(selection[i])

        # If initial selection was not correct answer
        # then contestant will win after swapping
        # Since list of available choices for the host
        # to remove will have only one element
        if len(available_choices) == 1:
            num_wins += 1

    return num_wins / n


def coin_toss(correct, selection):
    n = len(correct)
    num_wins = 0

    for i in range(n):
        available_choices = [0, 1, 2]

        # Removing a wrong answer from available choices
        available_choices.remove(correct[i])

        if available_choices.__contains__(selection[i]):
            available_choices.remove(selection[i])

        available_choices.remove(custom_random.choice(available_choices))
        available_choices.append(correct[i])  # Adding the correct choice back

        if not available_choices.__contains__(selection[i]):
            available_choices.append(selection[i])

        # Randomly selecting one door from the remaining two
        new_selection = custom_random.choice(available_choices)

        if new_selection == correct[i]:
            num_wins += 1

    return num_wins / n


NUM_SIMULATIONS = 10000
correct = custom_random.randint(low=0, high=3, size=NUM_SIMULATIONS)
selection = custom_random.randint(low=0, high=3, size=NUM_SIMULATIONS)

print('Never Swap:', never_swap(correct, selection))
print('Always Swap:', always_swap(correct, selection))
print('Coin Toss:', coin_toss(correct, selection))
```

2 Question 2

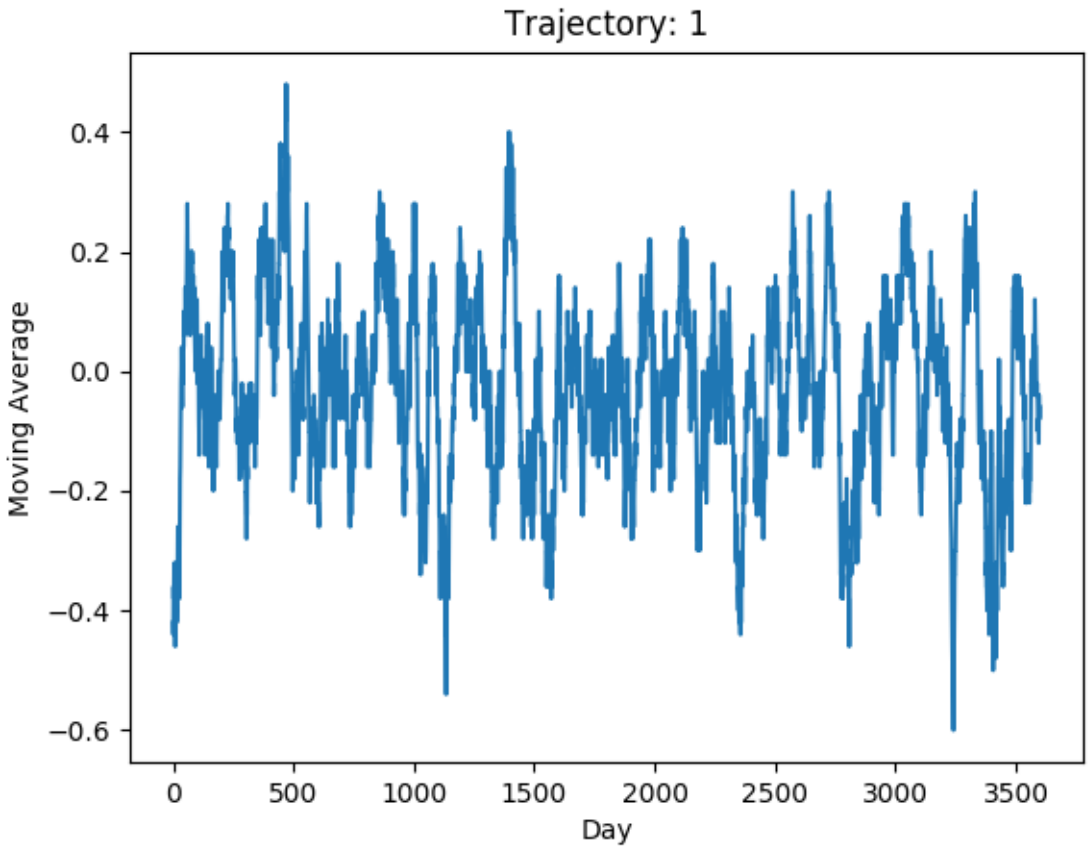
Simple moving average is taken using 50 consecutive data points. Data for each day is randomly taken from set S.

2.1 Pseudo Code for Moving Average

- moving_average(data, points_per_avg)
- n = length(a)
- for i in 0 to n - points_per_avg
- avg_data[i] = mean(data[i to (i + points_per_avg)])
- return avg_data

2.2 Plot for the Trajectory

All the trajectories are plotted. One of them is presented below.



2.3 Code

stock_price.py

```
import numpy as np
import matplotlib.pyplot as plt
import custom_random


DAYS_PER_YEAR = 365
```

```
NUM_YEARS = 10
NUM_DATA_POINTS = DAYS_PER_YEAR * NUM_YEARS
NUM_TRAJECTORIES = 10
S = np.array([-2, -1, 0, 1, 2])


# Returns array which stores the movivng average of the data
def simple_moving_avg(data, points_per_avg):
    n = len(data)
    avg_data = np.zeros(n-points_per_avg)

    for i in range(n-points_per_avg):
        avg_data[i] = np.mean(data[i:i+points_per_avg])

    return avg_data


for i in range(NUM_TRAJECTORIES):
    points_per_avg = 50

    # Randomly choosing points from S
    data = S[custom_random.randint(low=0, high=5, size=NUM_DATA_POINTS)]
    avg_data = simple_moving_avg(data, points_per_avg)

    # Plotting each trajectory
    plt.figure()
    plt.title('Trajectory: ' + str(i+1))
    plt.xlabel('Day')
    plt.ylabel('Moving Average')
    plt.plot(avg_data)

plt.show()
```

3 Question 3

The program is divided into 3 main parts:

3.1 Main

It is the entry point to the program. It initializes the simulation by creating an instance of GridDrawer.

3.2 GridDrawer

Initializes the OpenGL window and handles the drawing as the game iterates.

3.3 GameOfLife

This is used to create the Game of Life instance. An $N \times N$ grid is created. It is a boolean grid whose cells store *true* if an organism is present in it, and *false* otherwise. A certian number of seeds are randomly arranged in the grid that act as the initial configuration. *iterate()* function determines the new configuration of the grid based on the rules, and return the new grid which is used in GridDrawer.

3.3.1 Algorithm

- iterate(grid, N)

- newGrid[][] = boolean[N][N]
- for (i, j) in (0, 0) to (N-1, N-1)
- nCount = getNCount(grid, i, j) # Neighbour count of grid[i][j]
- if (nCount = 3)
- newGrid[i][j] = true # Organism is born
- else if(nCount = 2)
- newGrid[i][j] = grid[i][j] # State does not change
- else
- newGrid[i][j] = false # Organism dies
- return newGrid

3.4 Code

Main.java

```
public class Main {

    public static void main(String[] args) {

        GridDrawer gridDrawer = new GridDrawer();
        gridDrawer.draw();
    }
}
```

GridDrawer.java

```
import com.jogamp.opengl.*;
import com.jogamp.opengl.awt.GLCanvas;
import com.jogamp.opengl.util.FPSAnimator;

import javax.swing.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseWheelEvent;

public class GridDrawer implements GLEventListener {

    // Parameters for dragging window functionality
    // And controlling zoom
    private float DRAG_FACTOR = 500.0f;
    private float ZOOM_FACTOR = 1.1f;
    private int MAX_DRAG = 50;

    private float zoom = 2.0f;
    private int prevScreenX = 0;
```

```
private int prevScreenY = 0;
private int curScreenX = 0;
private int curScreenY = 0;

private float windowX = 0.0f;
private float windowY = 0.0f;

private GameOfLife game;
private int gridSize;
private float cellLength;
private int frameIndex;


public GridDrawer() {
    game = new GameOfLife(); // Initializing Game of Life

    gridSize = GameOfLife.getGridSize();
    cellLength = GameOfLife.getCellLength();
    frameIndex = 0;
}

public void draw() {
    // Setting up the OpenGL window
    final GLProfile profile = GLProfile.get(GLProfile.GL2);
    GLCapabilities capabilities = new GLCapabilities(profile);
    final GLCanvas glcanvas = new GLCanvas(capabilities);

    glcanvas.addGLEventListener(this);
    glcanvas.setSize(800, 800);

    final JFrame frame = new JFrame ("Game of Life");
    frame.getContentPane().add(glcanvas);
    frame.setSize(frame.getContentPane().getPreferredSize());
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    frame.setVisible(true);

    glcanvas.addMouseWheelListener(new MouseAdapter() {
        @Override
        public void mouseWheelMoved(MouseWheelEvent e) {
            zoom *= e.getWheelRotation() < 0 ? ZOOM_FACTOR : 1 / ZOOM_FACTOR;
        }
    });

    glcanvas.addMouseMotionListener(new MouseAdapter() {
        @Override
        public void mouseDragged(MouseEvent e) {
            curScreenX = e.getX();
            curScreenY = e.getY();

            if (Math.abs(curScreenX - prevScreenX) > MAX_DRAG
                    || Math.abs(curScreenY - prevScreenY) > MAX_DRAG) {
                prevScreenX = curScreenX;
                prevScreenY = curScreenY;
                return;
```

```
            }

            windowX += (curScreenX - prevScreenX) / DRAG_FACTOR / zoom;
            windowY += -(curScreenY - prevScreenY) / DRAG_FACTOR / zoom;
            prevScreenX = curScreenX;
            prevScreenY = curScreenY;
        }
    });

    // Frequency of the simulation is taken to be 10
    FPSAnimator animator = new FPSAnimator(glcanvas, 10, true);
    animator.start();
}

@Override
public void init(GLAutoDrawable glAutoDrawable) {
}

@Override
public void dispose(GLAutoDrawable glAutoDrawable) {
}

@Override
public void display(GLAutoDrawable glAutoDrawable) {
    final GL2 gl = glAutoDrawable.getGL().getGL2();
    gl.glClear (GL2.GL_COLOR_BUFFER_BIT);

    gl.glLoadIdentity();
    gl.glScalef(zoom, zoom, 1.0f);
    gl.glTranslatef(-0.5f + windowX, -0.5f + windowY, 0);

    boolean grid[][] = game.iterate(); // Getting the next grid data

    // Drawing the grid
    for (int i = 0; i < gridSize; ++i) {
        for (int j = 0; j < gridSize; ++j) {
            if (grid[i][j]) {
                gl.glBegin(GL2.GL_QUADS);

                gl.glVertex2f(j* cellLength, i* cellLength);
                gl.glVertex2f((j+1)* cellLength, i* cellLength);
                gl.glVertex2f((j+1)* cellLength, (i+1)* cellLength);
                gl.glVertex2f(j* cellLength, (i+1)* cellLength);

                gl.glEnd();
            }
        }
    }

    gl.glFlush();

    System.out.println("Frame Index: " + ++frameIndex);
}
```

```
    @Override
    public void reshape(GLAutoDrawable glAutoDrawable,
                        int i, int i1, int i2, int i3) {
        final GL2 gl = glAutoDrawable.getGL().getGL2();

        gl.glLoadIdentity();
        gl.glScalef(zoom, zoom, 1.0f);
        gl.glTranslatef(-0.5f + windowX, -0.5f + windowX, 0);
    }
}
```

GameOfLife.java

```
import java.util.Random;

public class GameOfLife {

    // Setting up the parameters of the Game of Life
    private static final int GRID_SIZE = 250;
    private static final int NUM_SEEDS = 8000;
    private static final float CELL_LENGTH = 1.0f / GRID_SIZE;

    private boolean[][] grid;


    public GameOfLife() {
        grid = new boolean[GRID_SIZE][GRID_SIZE];
        generateSeeds();
    }


    // Puts seeds on random locations in the grid
    private void generateSeeds() {
        Random random = new Random();

        for (int i = 0; i < NUM_SEEDS; i++) {
            int xi = random.nextInt(GRID_SIZE);
            int yi = random.nextInt(GRID_SIZE);
            grid[yi][xi] = true;
        }
    }


    // Returns the next iteration of the grid
    public boolean[][] iterate() {

        boolean newGrid[][] = new boolean[GRID_SIZE][GRID_SIZE];

        // Copying old grid to new one
```

```
        for (int i = 0; i < GRID_SIZE; ++i) {
            System.arraycopy(grid[i], 0, newGrid[i], 0, GRID_SIZE);
        }

        // Applying the rules to each element of the grid
        for (int i = 0; i < GRID_SIZE; ++i) {
            for (int j = 0; j < GRID_SIZE; ++j) {
                int neighbourCount = getNeighbourCount(i, j);

                // Organism is born
                if (neighbourCount == 3) {
                    newGrid[i][j] = true;
                    continue;
                }

                // Cell does not change its state
                if (neighbourCount == 2)
                    continue;

                // For the remaining conditions
                // (neighbourCount > 3 || neighbourCount < 2)
                // The organism dies
                newGrid[i][j] = false;
            }
        }

        grid = newGrid; // Updating the grid

        return newGrid;
    }


    // Returns the number of neighbours for a cell location
    private int getNeighbourCount(int i, int j) {
        int neighbourCount = 0;

        for (int k = -1; k <= 1; ++k) {
            for (int l = -1; l <= 1; l++) {
                int ii = i + k;
                int jj = j + l;

                // Boundary conditions
                if (ii == -1 || jj == -1 || ii == GRID_SIZE || jj == GRID_SIZE)
                    continue;

                // Leaving the cell itself
                if (k == 0 && l == 0)
                    continue;

                if (grid[ii][jj])
                    ++neighbourCount;
            }
        }
```

```
        return neighbourCount;
    }


    public static int getGridSize() {
        return GRID_SIZE;
    }


    public static float getCellLength() {
        return CELL_LENGTH;
    }
}
```

4 Question 4

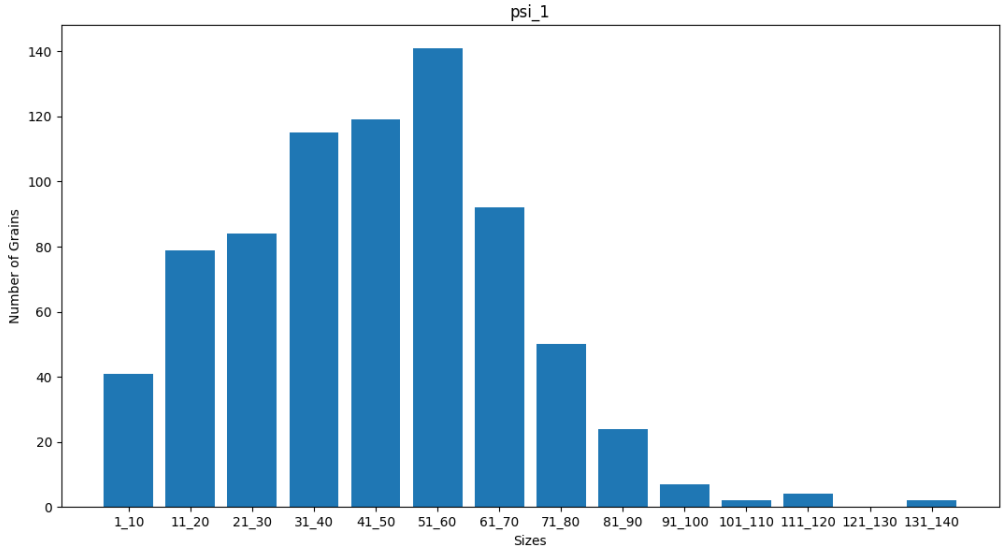
Instead of iterating over all the 512 columns, only 50 columns are randomly chosen and the grain sizes are determined by traversing through each of them. The distribution is plotted and average size is taken as the size of the grains.

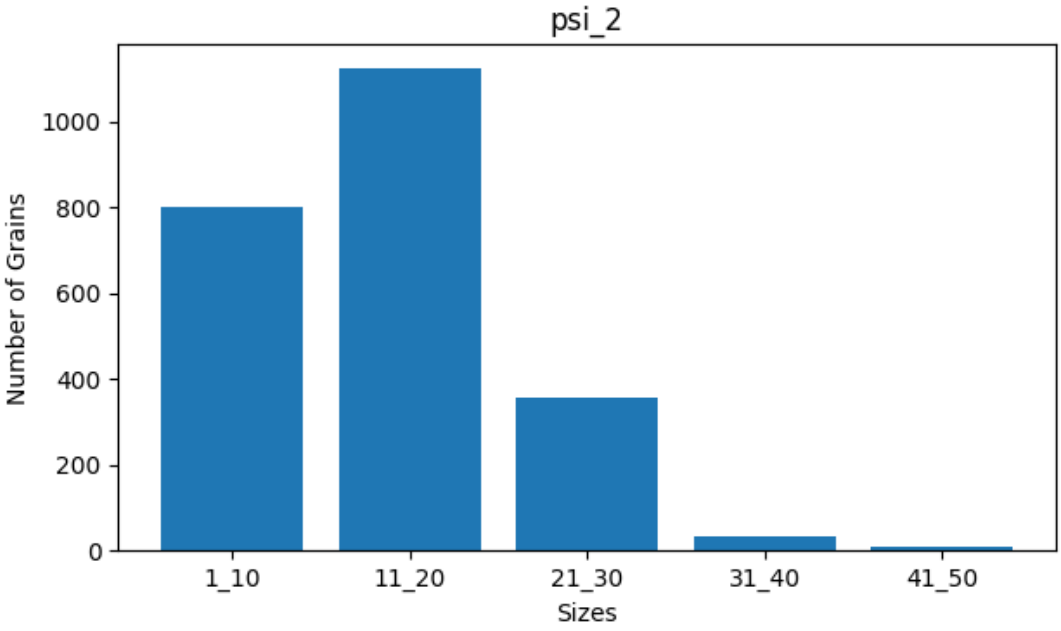
4.1 Pseudo Code

- column_indices = randint(0, 512, 50)
- grain_sizes = []
- for i in column_indices
- append determined sizes in column[i] to grain_sizes list
- mean_size = mean(grain_sizes)
- frequency = 1 / (mean_size × mean_size) # grains per unit area
- divide grain_sizes into ranges
- plot the distribution

4.2 Results

	Mean Size (units)	Frequency (grains per unit area)
psi_1.dat	45.67	4.7 e-4
psi_2.dat	14.09	5.0 e-3





4.3 Code

grain.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import custom_random


def get_grain_sizes(psi):
    sizes = []

    # Iterating through 50 columns chosen randomly
    for i in custom_random.randint(low=0, high=512, size=50):
        column = psi[i]
        start_j = 0  # Stores the index of start of a grain
```

```
        end_j = 0    # Stores the index of end of a grain

        for j in range(1, 512):

            # Grain starts
            if column[j] - column[j - 1] > 0.1:
                start_j = j

            # Grain ends
            elif column[j] - column[j - 1] < -0.1:
                end_j = j
                sizes.append(end_j - start_j)

    return sizes


# Divides the grain sizes according to the range they lie in
def get_distribution(sizes, bin_size=10):
    dist = init_distribution_dict(bin_size, max(sizes))

    for s in sizes:
        rs = get_elem_range_str(s, bin_size)
        dist[rs] += 1

    return dist


# Initializes all entries of distribution dictionary to 0
def init_distribution_dict(bin_size, max_element):
    dist = {}

    for r in range(1, max_element, bin_size):
        dist[get_elem_range_str(r, bin_size)] = 0

    return dist


# Returns range string for an element
# Eg. if bin_size = 10, elem = 15, return value is '11_20'
def get_elem_range_str(elem, bin_size):
    i = 1
    # Finding range in which the element lies
    while bin_size*i < elem:
        i += 1

    rs = str(bin_size*(i-1) + 1) + '_' + str(bin_size*i)

    return rs


def process_psi(file_name):
    psi = pd.read_csv(filepath_or_buffer=file_name,
                      delimiter='  ', engine='python',
                      header=None)
```

```
    grain_sizes = get_grain_sizes(psi)
    dist = get_distribution(grain_sizes)

    mean_size = np.mean(grain_sizes)
    frequency = 1 / (mean_size**2)
    print(file_name, 'Mean Size: ', mean_size)
    print(file_name, 'Frequency: ', frequency)
    plt.figure()
    plt.bar(list(dist.keys()), list(dist.values()))
    plt.title(file_name[0:file_name.rindex('.')])
    plt.xlabel('Sizes')
    plt.ylabel('Number of Grains')


process_psi('psi_1.dat')
process_psi('psi_2.dat')
plt.show()
```