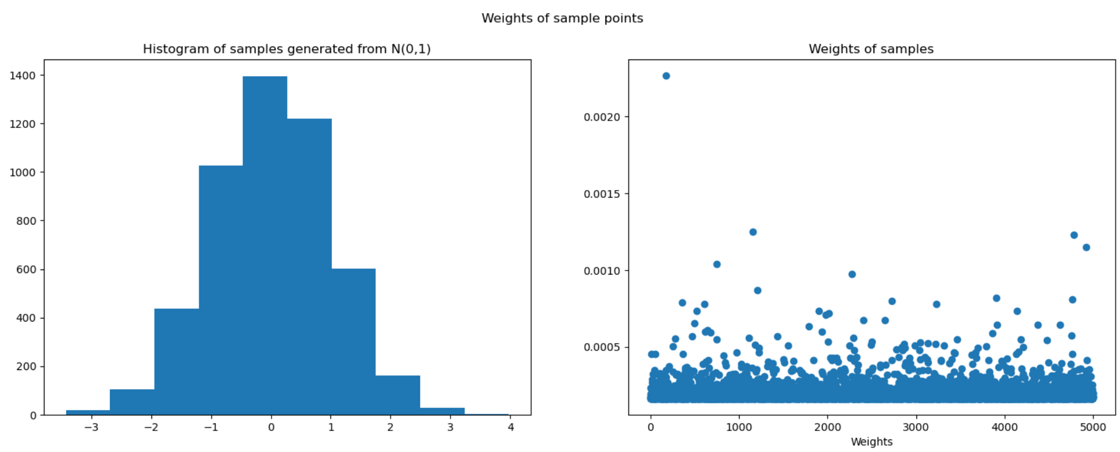
Fall 2022: Monte Carlo Methods
Homework 3

NAME: Utkarsh Khandelwal
Net Id: uk2051

Exercise 28

We are given a target distribution $N(0, \sigma^2)$ and we are using the Standard Gaussian $N(0, 1)$ to generate samples. I implemented the mentioned re-sampling methods: 'Multinomial, Bernoulli and Systematic. Below is the sample image showing the weights of the points sampled from $N(0, 1)$.



We can also see that these re-sampling methods are still producing the unbiased estimators. Below is the mean computed using

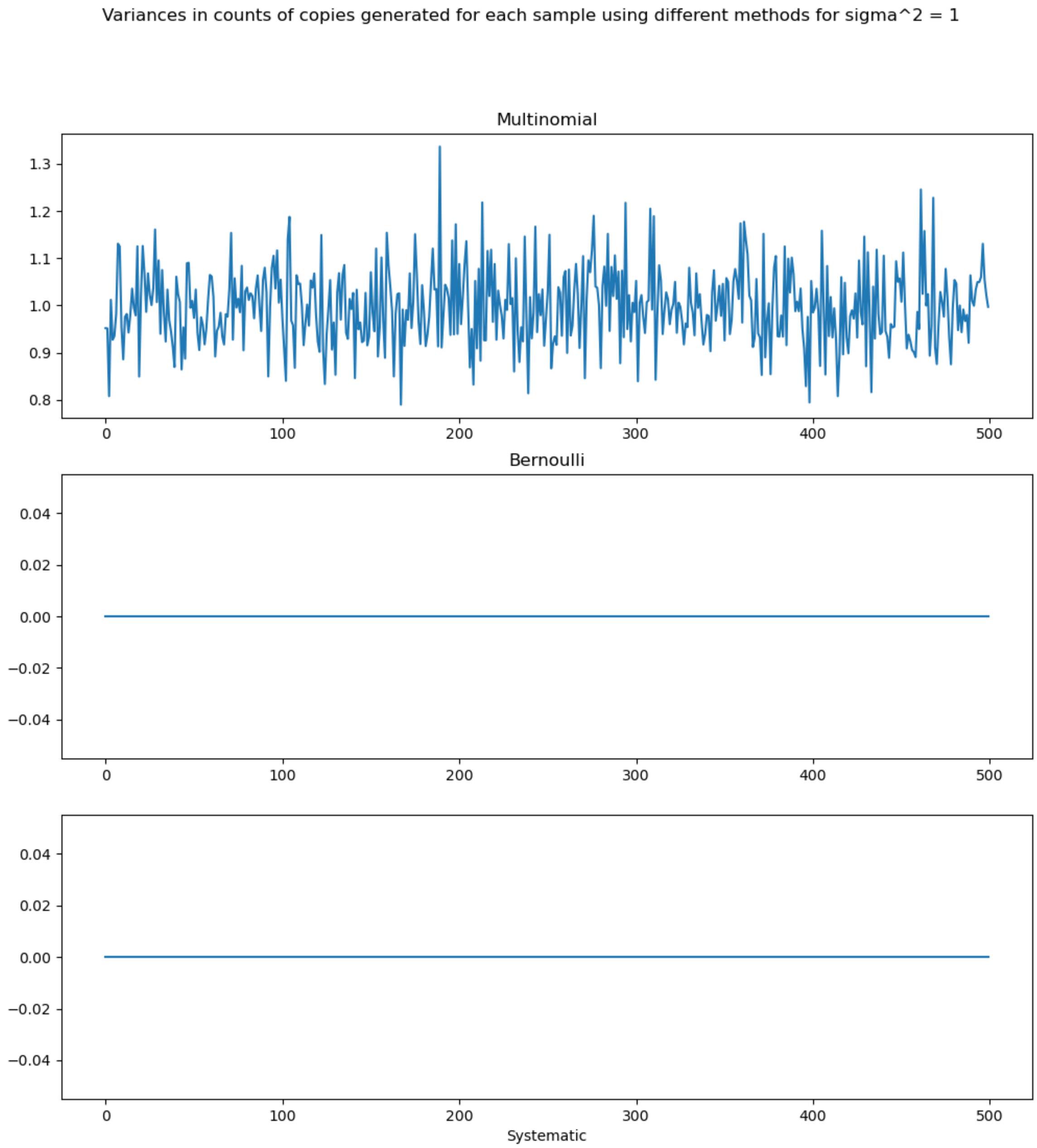
$$E[x] = \sum_{i=1}^{k} x^{(i)} w^{(i)}$$

```
/Users/utkarsh/.conda/envs/PyTorchTesting/bin/
Multinomial, mean:  0.007095422624287677
Bernaulli, mean:  -0.03118538840362053
Systematic, mean:  -0.02605670696395046
```

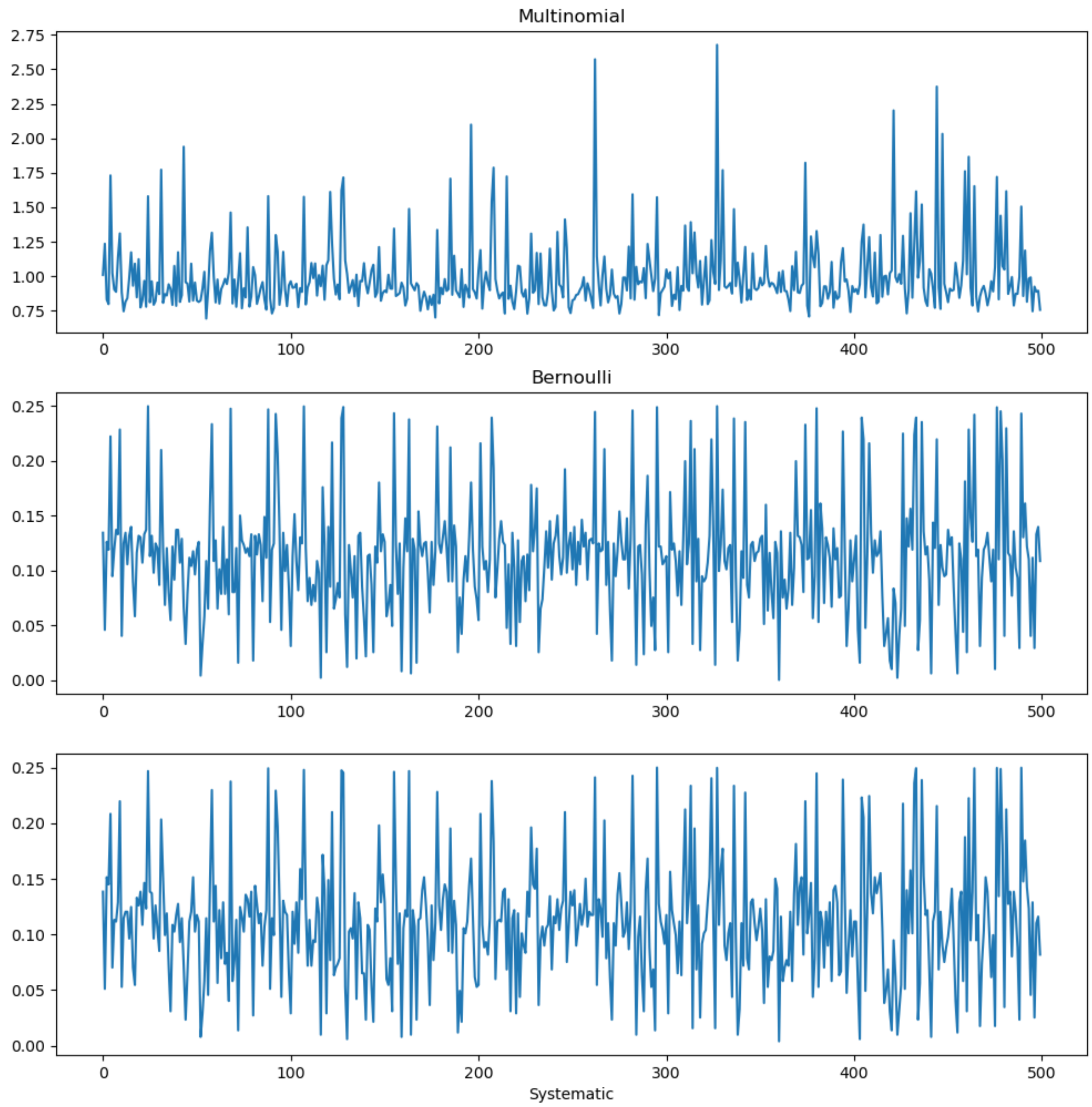
For the next part of the question, I generated five hundred sample points and computed weights for each sample respectively. So we have

$$X^{(i)}, w^{(i)} \text{ for } 0 \leq i < N$$

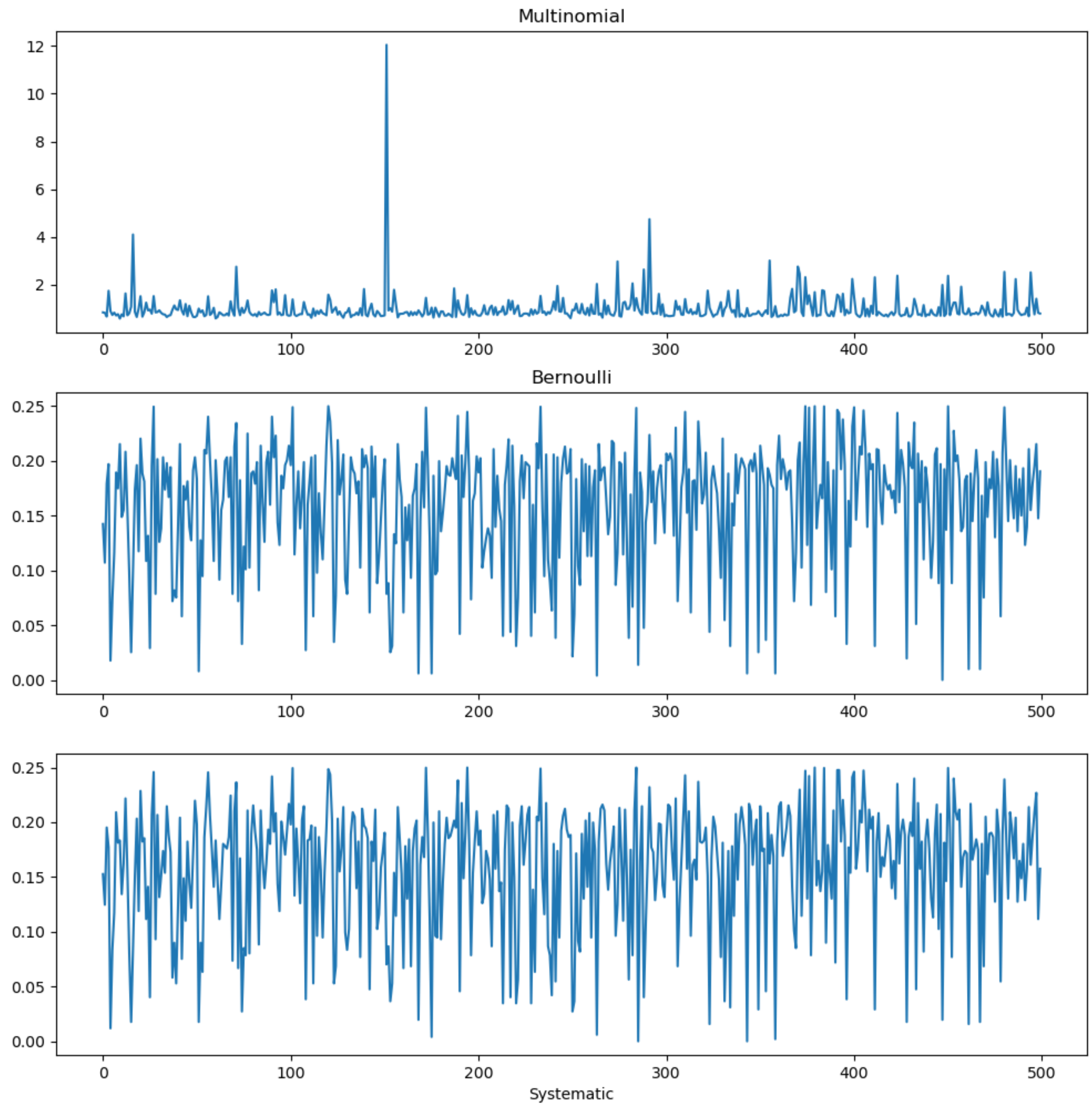
where $N = 500$ Next step was to re-sample the points with all the three methods and for re-sampling we will generate $N^{(k)}$ copies of the sample with weight $w^{(k)}$. Below are the figures showing variances that were computed normally for different values of σ^2.

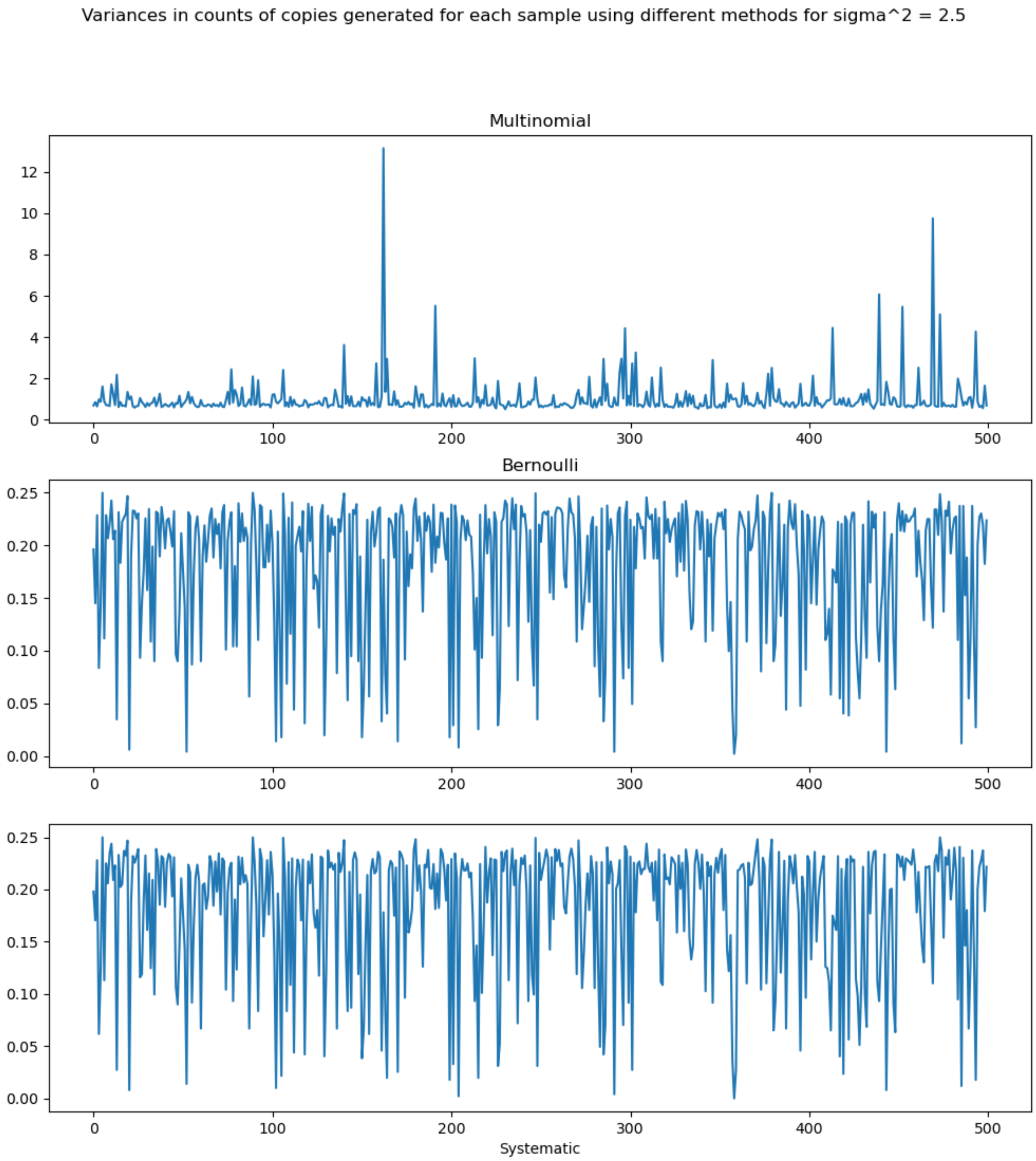

Variances in counts of copies generated for each sample using different methods for sigma^2 = 1
Multinomial
Bernoulli
Systematic

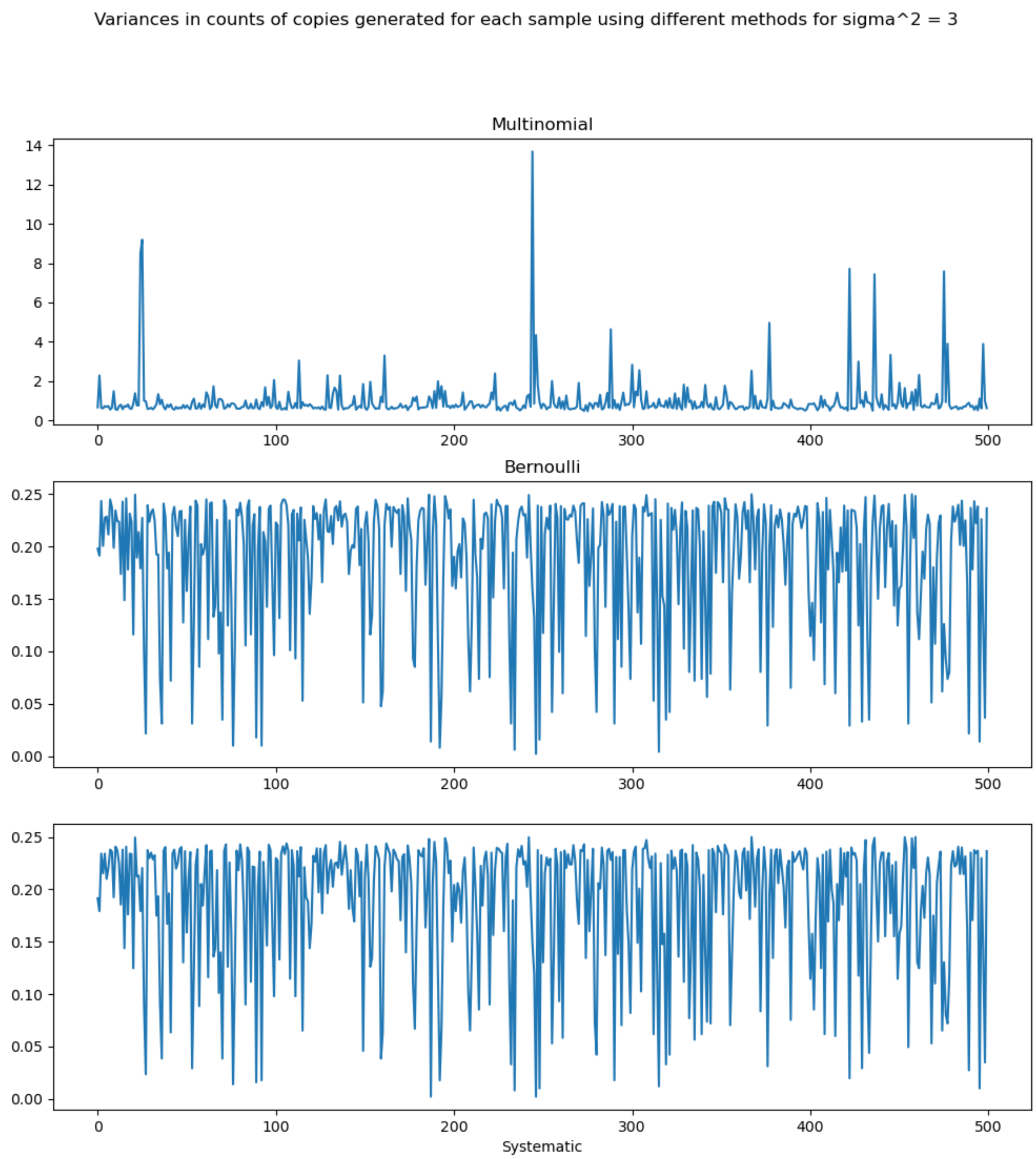
Variances in counts of copies generated for each sample using different methods for sigma^2 = 1.5



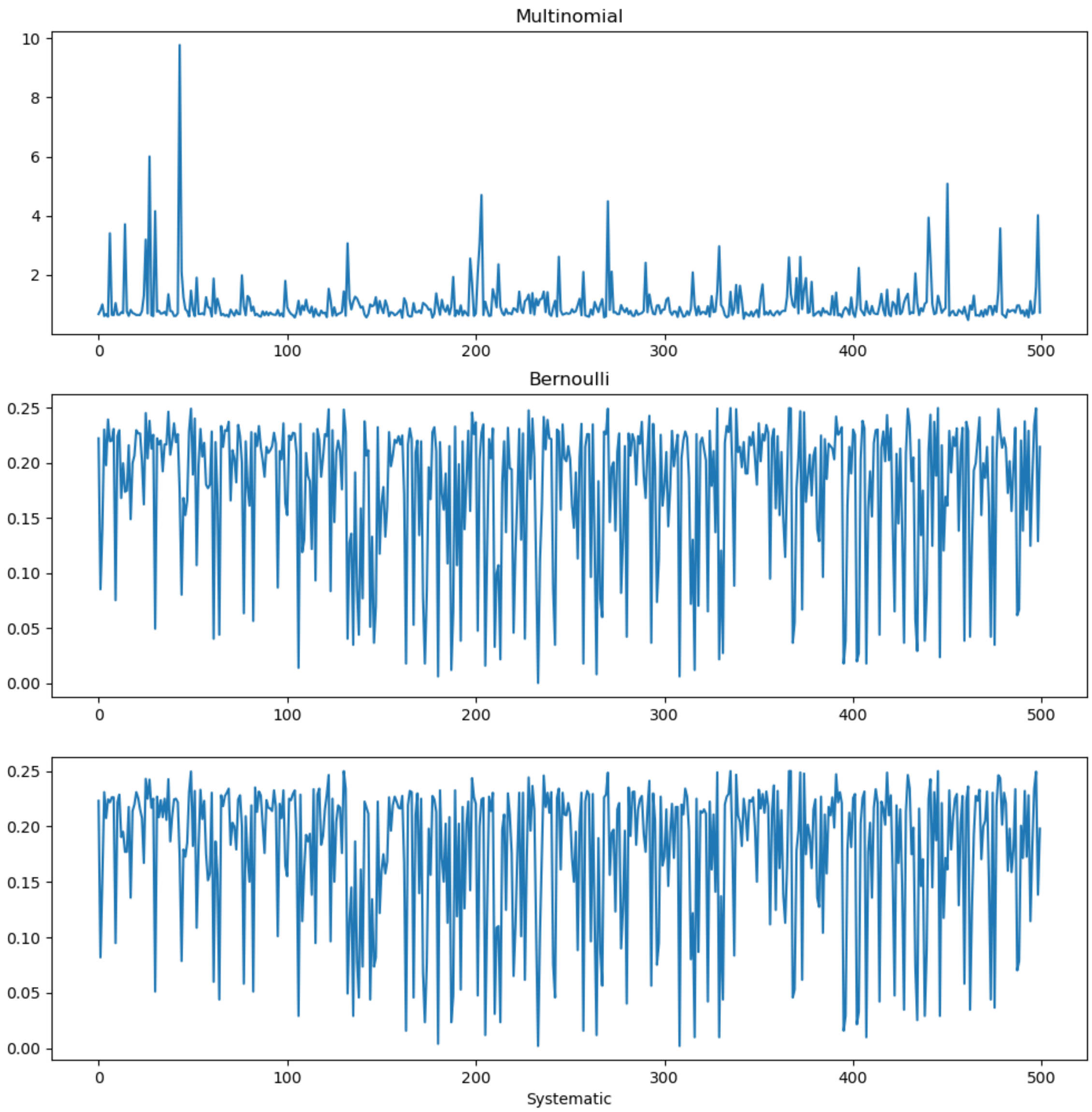
Variances in counts of copies generated for each sample using different methods for sigma^2 = 2



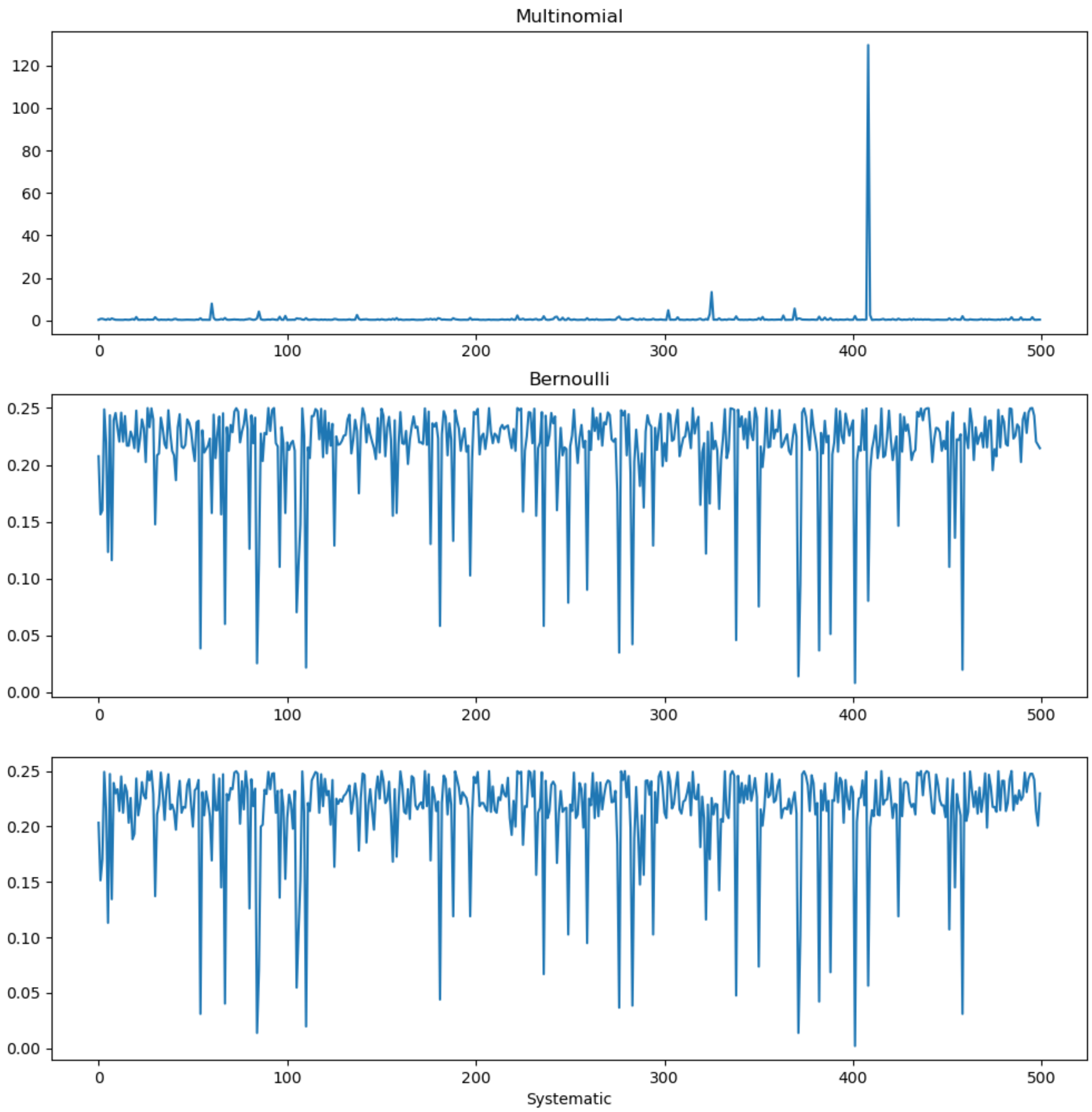

Variances in counts of copies generated for each sample using different methods for sigma^2 = 2.5
Multinomial
0
2
4
6
8
10
12
0
100
200
300
400
500
Bernoulli
0.00
0.05
0.10
0.15
0.20
0.25
0
100
200
300
400
500
0.00
0.05
0.10
0.15
0.20
0.25
0
100
200
300
400
500
Systematic

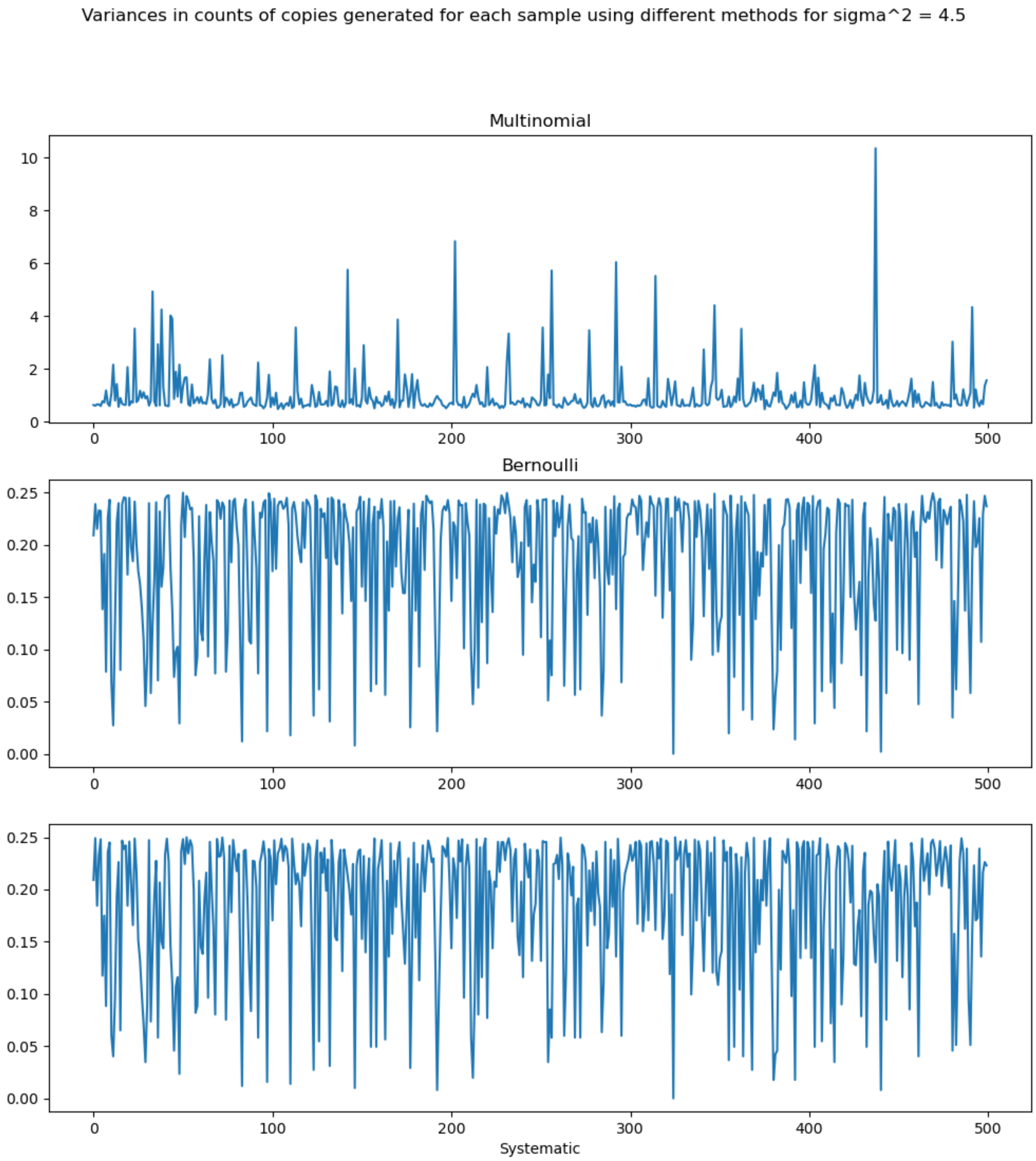

Variances in counts of copies generated for each sample using different methods for sigma^2 = 3
Multinomial
0
2
4
6
8
10
12
14
0
100
200
300
400
500
Bernoulli
0.00
0.05
0.10
0.15
0.20
0.25
Systematic

Variances in counts of copies generated for each sample using different methods for sigma^2 = 3.5



Variances in counts of copies generated for each sample using different methods for sigma^2 = 4





We can clearly observe that the variance for the Bernoulli and Systematic Re-sampling methods the variance is bounded between 0 and 0.25 while when using Multinomial for re-sampling the variance is sensitive with respect to the value of σ^2.

Below is the code snippet for the Multinomial Re-sampling method

```
def DoMultinomialResampling(x, w):
    m = len(w)
    x_resampled = np.zeros(0)
    w_bar = np.sum(w) / m
    probab_vector = w / (m * w_bar)  # Success probab of MultiNomialTrail
```

```
    copyCounts = np.random.multinomial(m, probab_vector)

    for i in range(m):
        cc = copyCounts[i]
        for j in range(cc):
            x_resampled = np.append(x_resampled, x[i])

    return (copyCounts, x_resampled)
```

Below is the code snippet for the Bernoulli Re-sampling method

```
def DoBernaulliResampling(x, w):
    m = len(w)
    w_bar = np.sum(w) / m
    x_resampled = np.zeros(0)

    copyCounts = np.zeros(m)
    for k in range(m):
        curr_copy_count = np.floor(m * w[k])
        u = np.random.uniform(0, 1)
        if (u < m * w[k] - curr_copy_count):
            curr_copy_count = curr_copy_count + 1
        else:
            curr_copy_count = curr_copy_count
        copyCounts[k] = (curr_copy_count.astype(np.int64))
    # print(copyCounts)
    copyCounts = copyCounts.astype(np.int64)
    # print(copyCounts)

    for i in range(m):
        cc = copyCounts[i]
        for j in range(cc):
            x_resampled = np.append(x_resampled, x[i])

    return (copyCounts, x_resampled)
```

And, below is the code snippet for Systematic Re-sampling method

```
def DoSystematicSampling(x, w):
    m = len(w)
    cumW = np.cumsum(w)

    u_s = []
    u = np.random.uniform(0, 1 / m)
    for i in range(1, m + 1):
        u_s = np.append(u_s, i / m - u)

    copyCounts = np.zeros(m)
    for k in range(1, m + 1):
        if (k == 1):
            sw = 0  # Starting weight
            fw = cumW[k - 1]  # Final weight
            # Count the num of us between sw and fw
            satisfies = [u for u in u_s if u >= sw and u < fw]
            copyCounts[k - 1] = len(satisfies)
        else:
            sw = cumW[k - 2]
            fw = cumW[k - 1]
            # Count the num of us between sw and fw
            satisfies = [u for u in u_s if u >= sw and u < fw]
```

```
            copyCounts[k - 1] = len(satisfies)

    x_resampled = np.zeros(0)
    w_bar = np.sum(w) / m

    copyCounts = copyCounts.astype(np.int64)

    for i in range(m):
        cc = copyCounts[i]
        # print(cc)
        for j in range(cc):
            x_resampled = np.append(x_resampled, x[i])
    return (copyCounts, x_resampled)
```

Exercise 29

In this exercise, I wrote routines to generate points from Self Avoiding Walks of length (n) for both with and without multinomial resampling.

Below is the snippet of the code. It takes are Boolean to switch on or off the re-sampling code.

```
def GetAvailableDirections(curr_x, curr_y, curr_points_in_saw):
  free_neighbors_count = 0;
  directions = [(1,0), (0,1), (-1, 0), (0,-1)]
  availableDirections = []
  for dir in range(4):
    del_x, del_y = directions[dir]
    n_x = curr_x + del_x
    n_y = curr_y + del_y
    if(not((n_x,n_y) in curr_points_in_saw)):
      free_neighbors_count +=1
      availableDirections.append((del_x, del_y))
  return free_neighbors_count, availableDirections

def GetNextPointInSAW(curr_x, curr_y, availableDirections):
  dirIndex = random.randint(0, len(availableDirections))
  del_x, del_y = availableDirections[dirIndex]
  new_x = curr_x + del_x  #New point x coordinate for this sample
  new_y = curr_y + del_y  #New point y coordinate for this sample
  return new_x, new_y

def GetCopyCounts(m, weights):
  mean = weights.mean()   #This is w bar, weight to be used after resampling
  probabVector = ((weights/(m * mean)).to_numpy()).astype(np.float64)
  #print(probabVector)
  copyCounts = np.random.multinomial(m, probabVector)
  return copyCounts

def GetSelfAvoidingWalk(m, d, resamplingOn = True):
  x_frame = pd.DataFrame(index=np.arange(m), columns=np.arange(d + 1))
  y_frame = pd.DataFrame(index=np.arange(m), columns=np.arange(d + 1))
  w_frame = pd.DataFrame(index=np.arange(m), columns=np.arange(d + 1))

  #Dimensions will run from 0 to d. Setting col 0 value to 0 for coords and 1 for weights
  x_frame[0] = 0
  y_frame[0] = 0
  w_frame[0] = 1
  #List of dictionaries for each sample. Dict will contain tuple of all the points in SAW.
  coords_set_main = []
```

```
  for i in range(m):
    coords_set_main.append({(0,0)})

  for dim_iter in range(d):                                    #All dimension
    for sample_iter in range(m):                                #All samples

      curr_x = x_frame.at[sample_iter,dim_iter]
      curr_y = y_frame.at[sample_iter,dim_iter]
      free_neighbors_count, availableDirections = GetAvailableDirections(curr_x, curr_y, coords_set_main[sample_iter])
      if(free_neighbors_count == 0):
        #print('Returning false')
        return False, [], [], []    #Return Status as of now. Not processing this any further

      new_x, new_y = GetNextPointInSAW(curr_x, curr_y, availableDirections)
      (coords_set_main[sample_iter]).add((new_x, new_y))  #Adding in the dictionary

      #Add it into the dataframes
      x_frame.at[sample_iter,dim_iter + 1] = new_x
      y_frame.at[sample_iter,dim_iter + 1] = new_y
      w_frame.at[sample_iter,dim_iter + 1] = w_frame.at[sample_iter,dim_iter] * free_neighbors_count

    #All samples computed for curr dim_iter and are stored in dim_iter + 1

    #Resampling code below
    if(dim_iter < d - 1 and resamplingOn):
      if(dim_iter == 1):
        print('Resampling On')
      copyCounts = GetCopyCounts(m, w_frame[dim_iter + 1])
      x_resampled = []
      y_resampled = []

      for i in range(m):
        cc = copyCounts[i]
        for j in range(cc):
          x_resampled.append(x_frame.loc[i])
          y_resampled.append(y_frame.loc[i])

      if(len(x_resampled) != m or len(y_resampled) != m):
        print('Size not equal')

      for i in range(m):
        x_frame.loc[i] = x_resampled[i]
        y_frame.loc[i] = y_resampled[i]

  #print('Code ran successfully')
  return True, x_frame, y_frame, w_frame
```

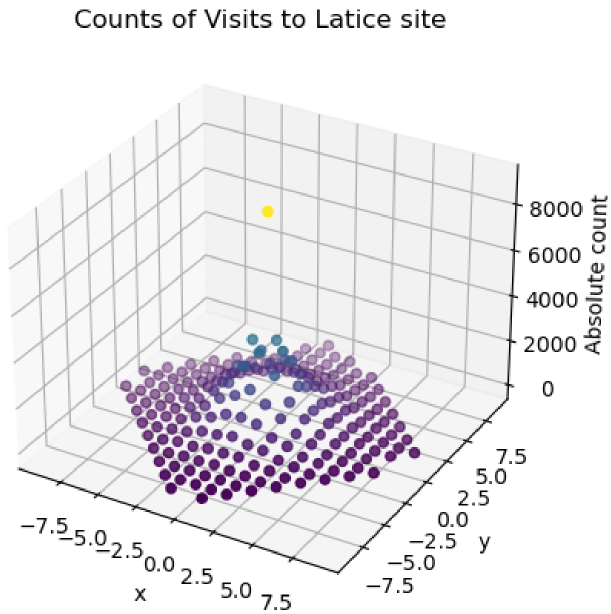
Nowe, for validating the code, I wrote sum wrapper and helper functions to find the expectation of the times, each Lattice point was visited for N SAWs of length d. Following is the output for few SAWs.

```
Expectation for d =  1
(x,y)       Count
-------  --------
(1, 0)   0.241676
(0, 1)   0.255604
(-1, 0)  0.245811
(0, -1)  0.25691
```

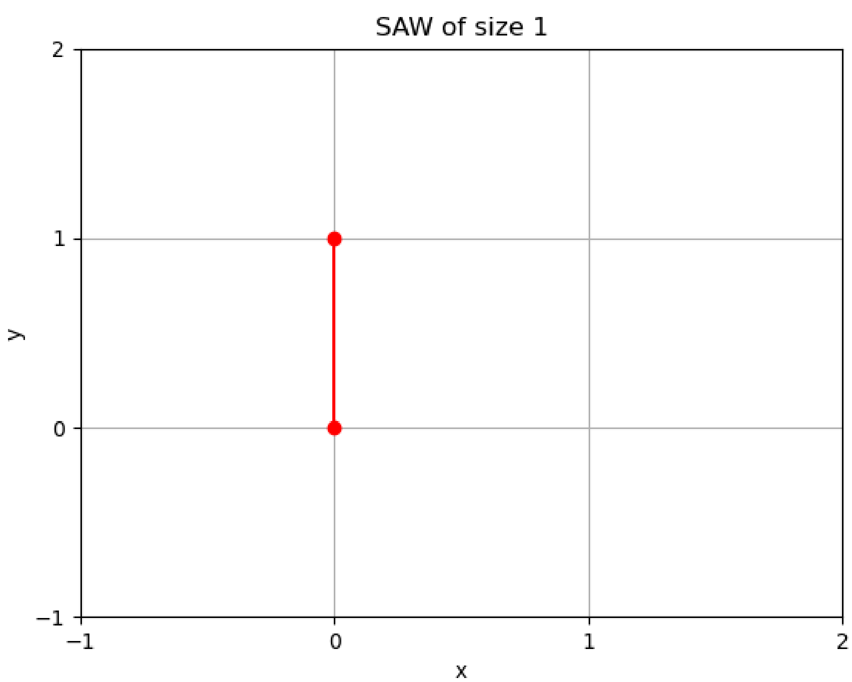
```
Expectation for d =  2
(x,y)         Count
--------  ---------
(1, -1)   0.165941
(0, 2)    0.0834603
(-1, 1)   0.174102
(0, -2)   0.089445
(-1, -1)  0.166812
(-2, 0)   0.0787813
(1, 1)    0.160936
(2, 0)    0.0805223
```

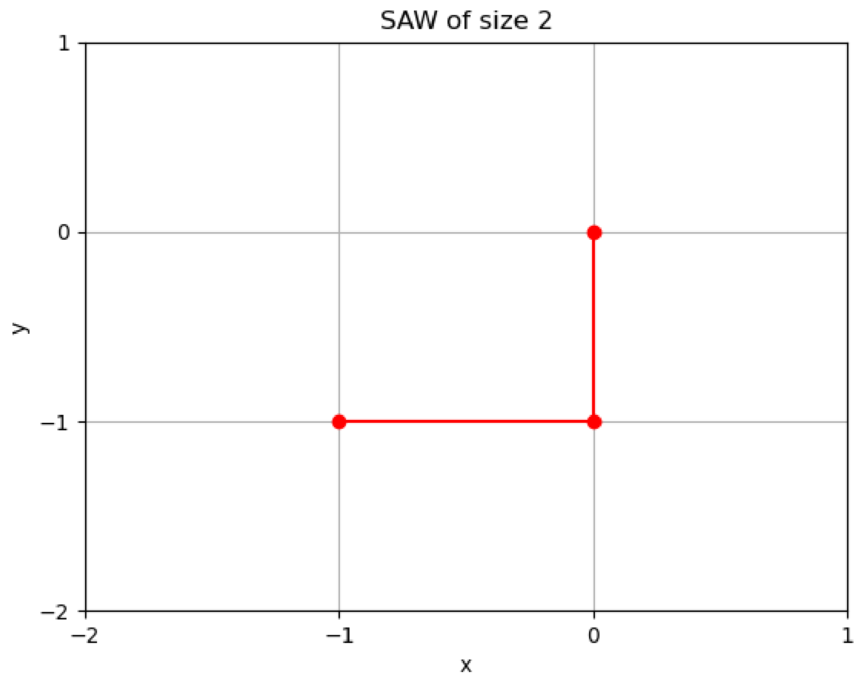
```
Expectation for d =  3
(x,y)         Count
--------  ---------
(0, -1)   0.0562568
(1, 2)    0.0808487
(-1, 0)   0.0568009
(-2, 1)   0.0831338
(2, -1)   0.0840044
(1, -2)   0.0858542
(-2, -1)  0.0828074
(0, 1)    0.0564744
(-1, 2)   0.0840044
(1, 0)    0.0535365
(2, 1)    0.0815016
(0, 3)    0.0275299
(3, 0)    0.0253536
(-3, 0)   0.0278564
(-1, -2)  0.0833515
(0, -3)   0.0306855
```

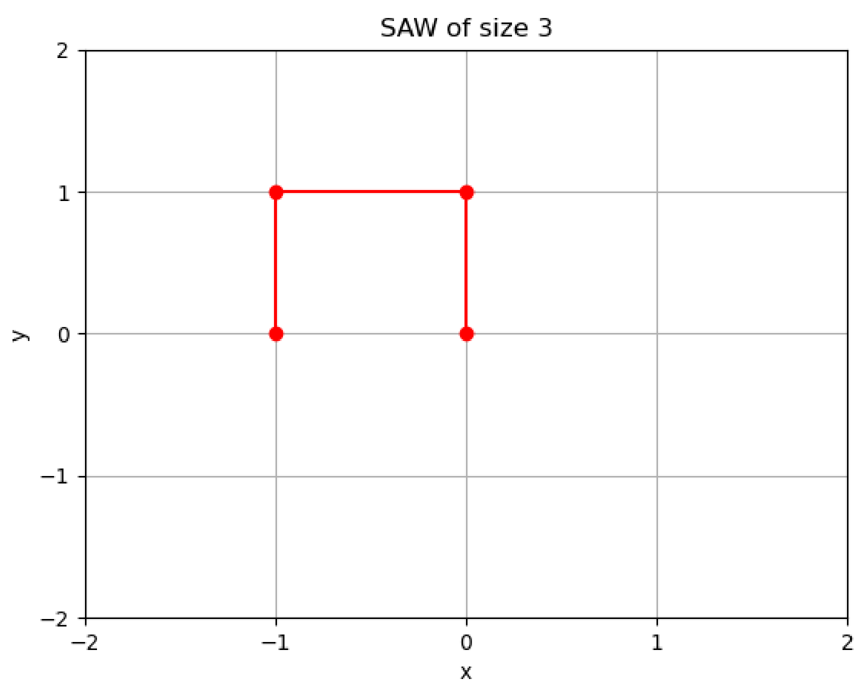
Also, plotted the variation of counts of visit of each Lattice point with the x and y coordinates. Following is the output obtained.

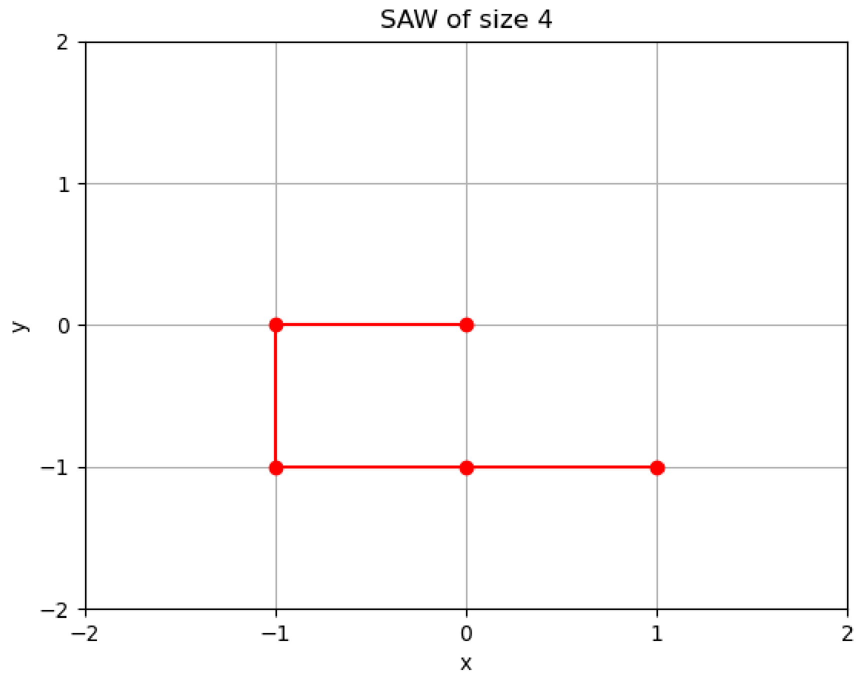


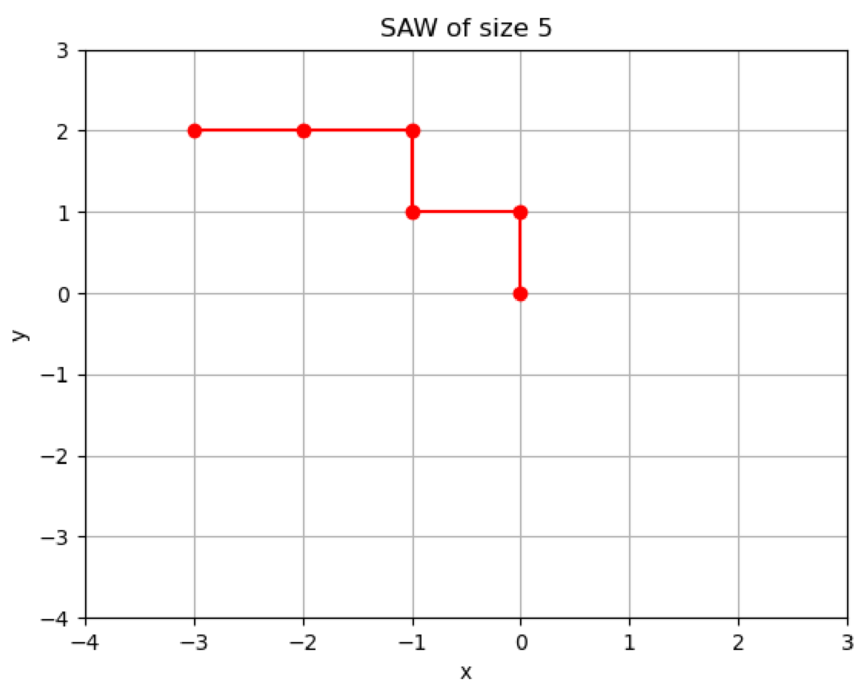
After checking that the method is valid, I produced SAW for each value d. Below are the generated paths.

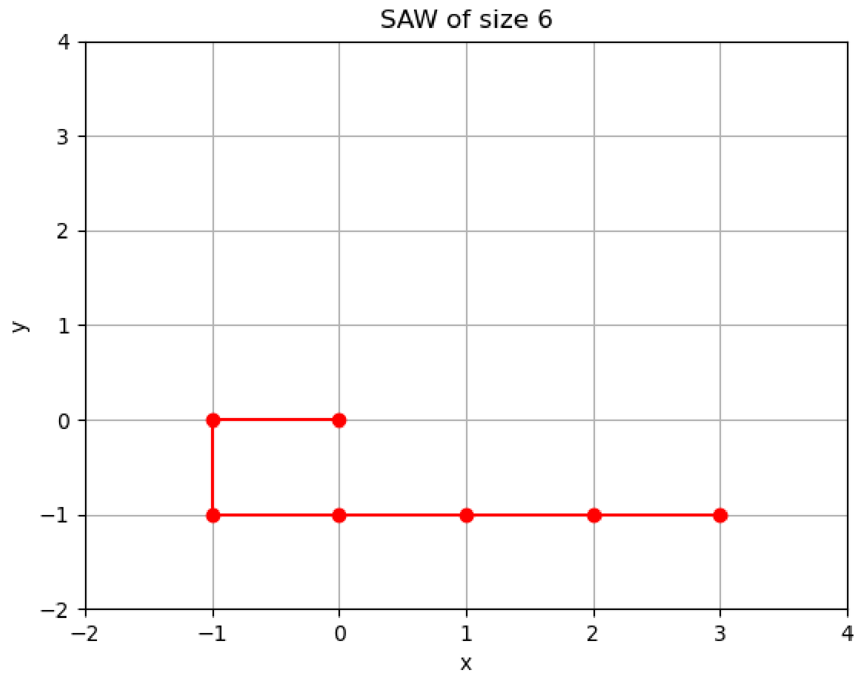


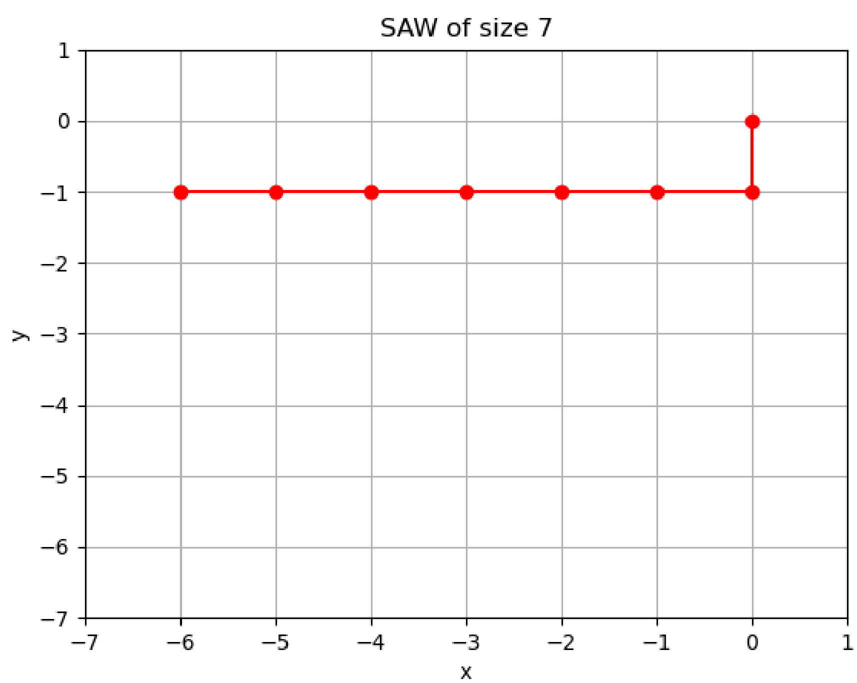

SAW of size 2
x
y
−2
−1
0
1

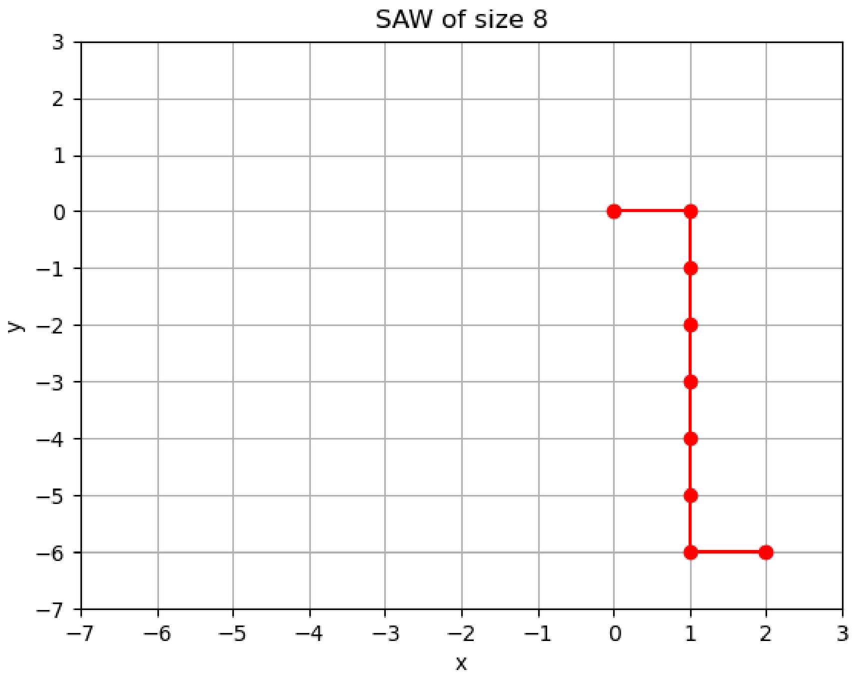

SAW of size 3
x
y
−2
−1
0
1
2

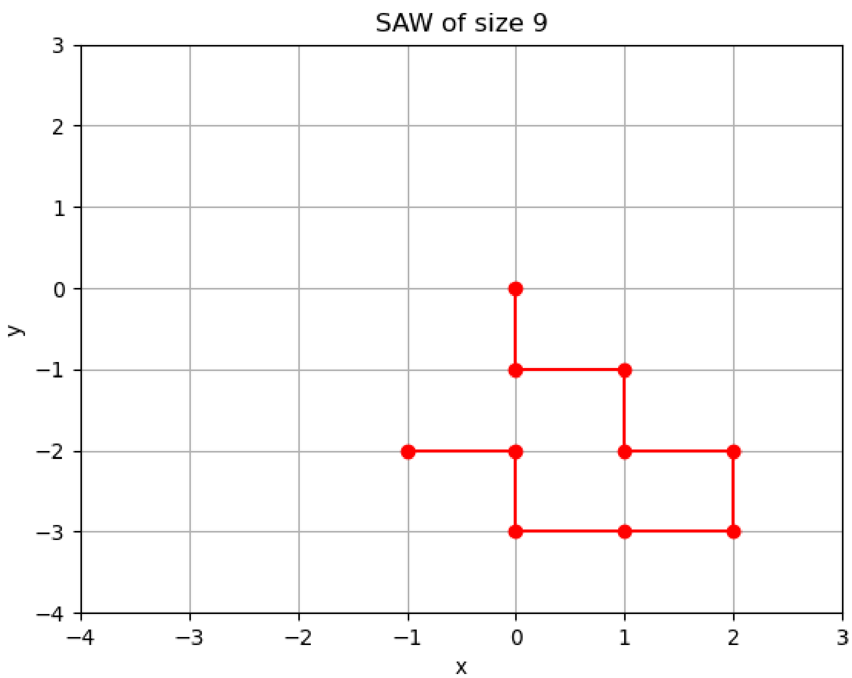

SAW of size 4
y
x

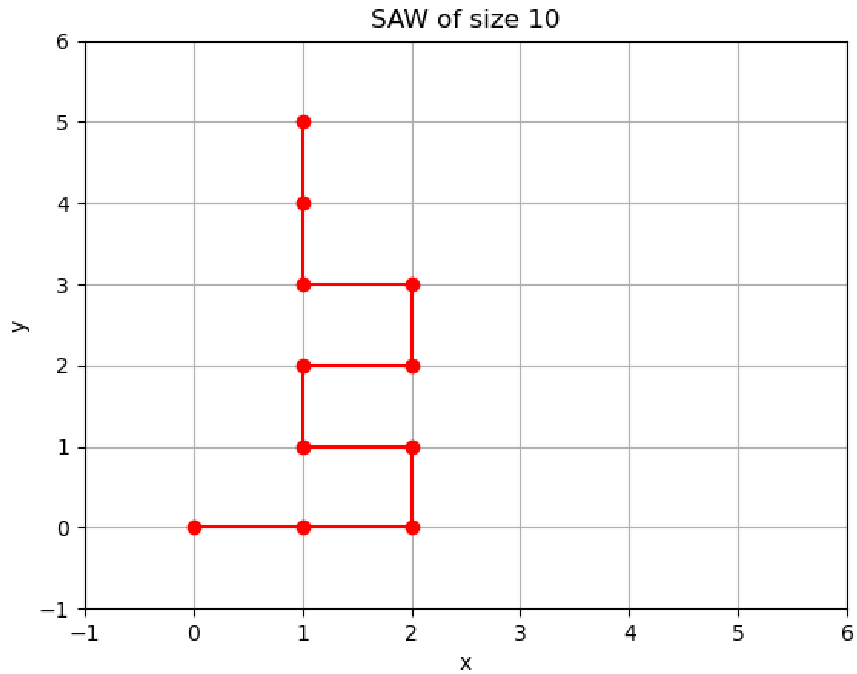

SAW of size 5
y
x

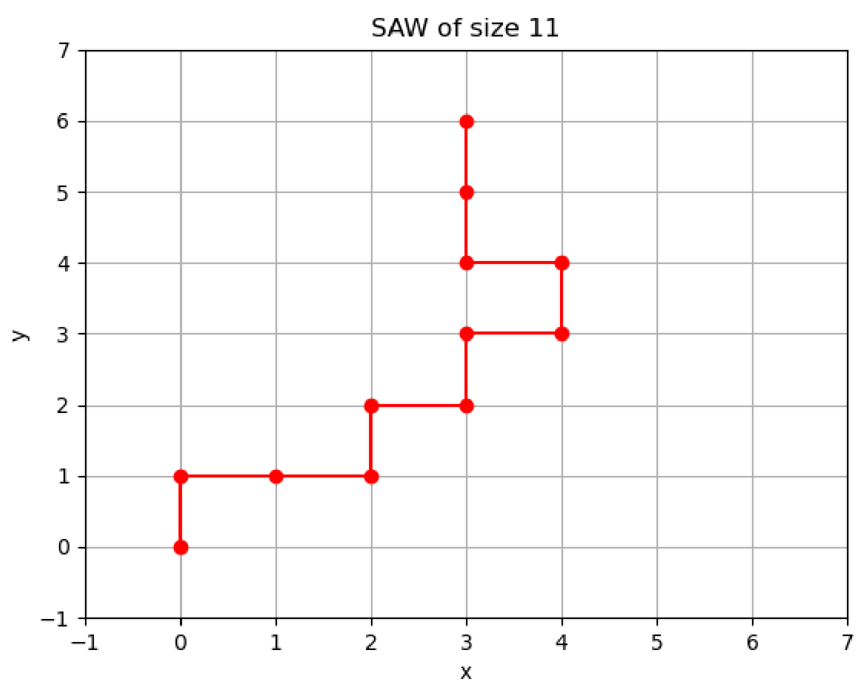

SAW of size 6
4
3
2
1
0
−1
−2
y
−2
−1
0
1
2
3
4
x

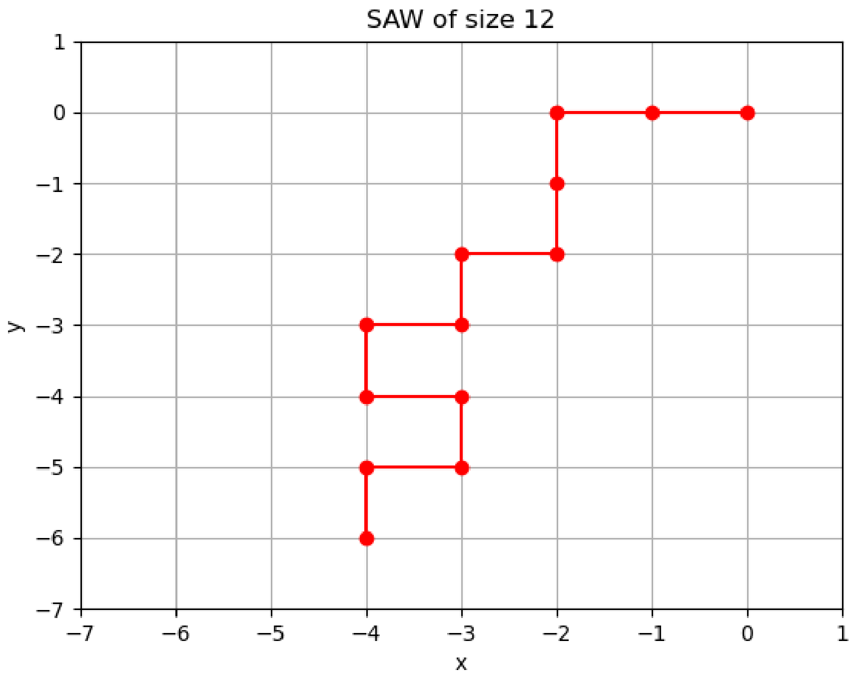

SAW of size 7
1
0
−1
−2
−3
−4
−5
−6
−7
y
−7
−6
−5
−4
−3
−2
−1
0
1
x

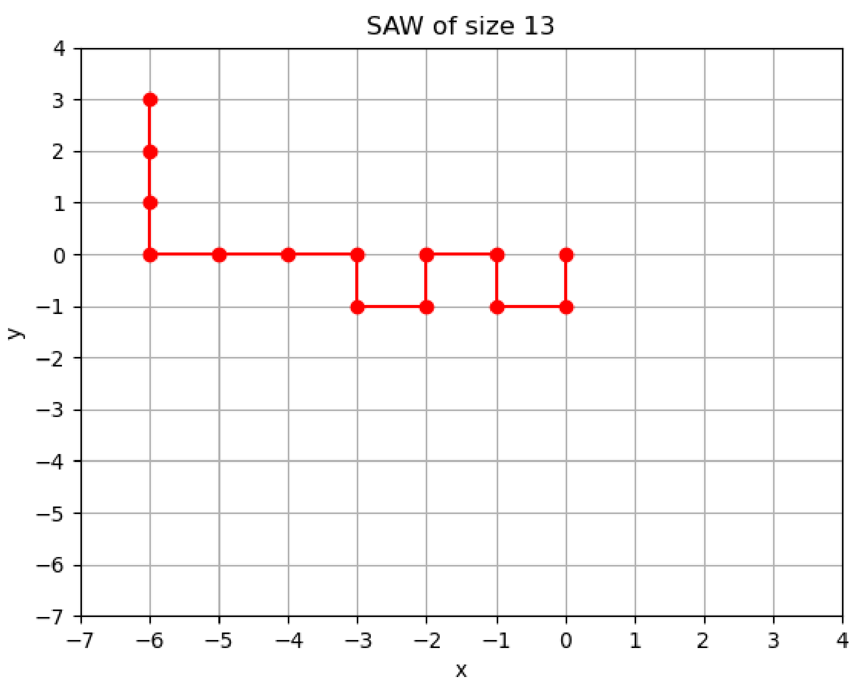

SAW of size 8
y
x
3
2
1
0
−1
−2
−3
−4
−5
−6
−7
−7
−6
−5
−4
−3
−2
−1
0
1
2
3

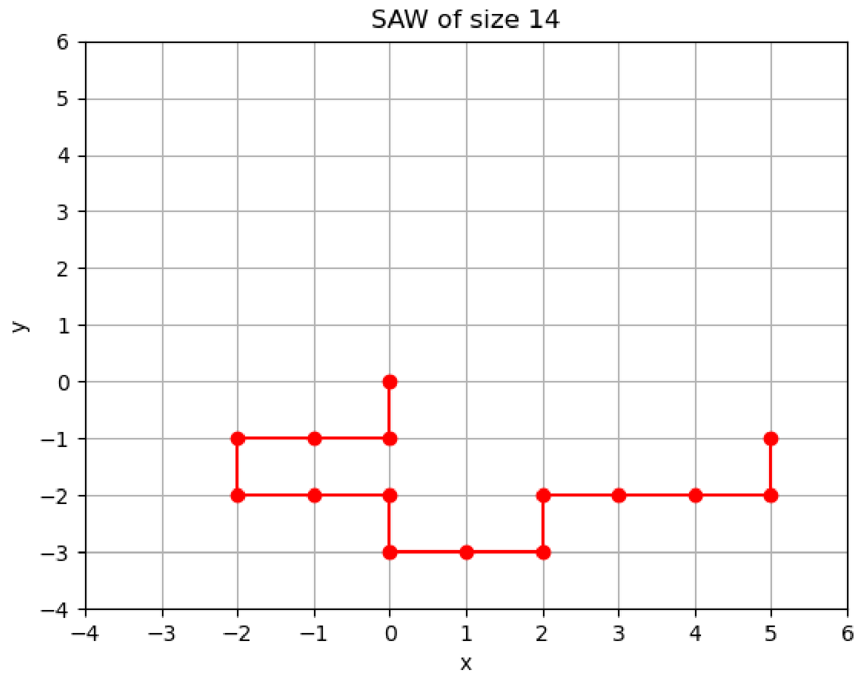

SAW of size 9
y
x
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3

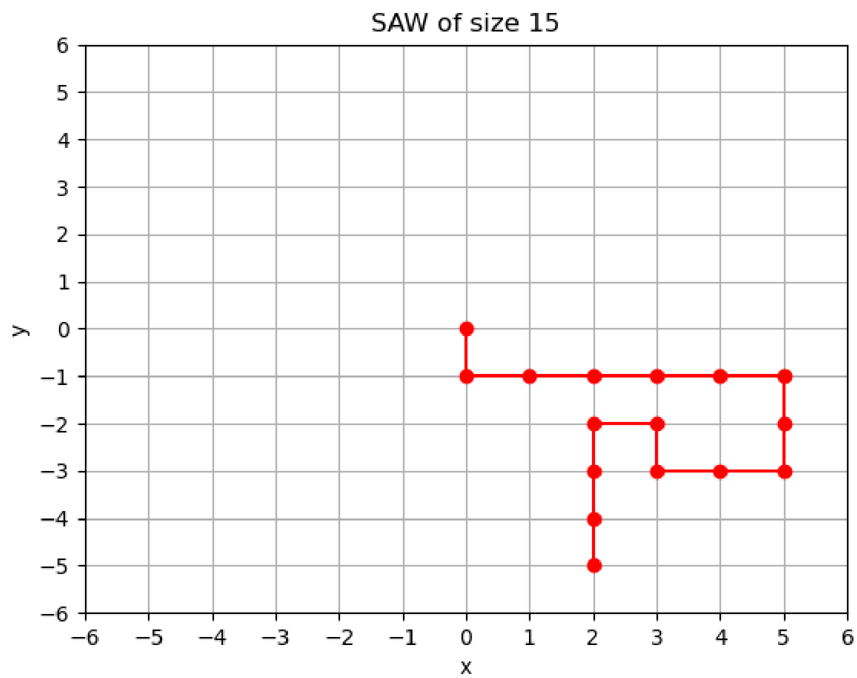

SAW of size 10
y
x

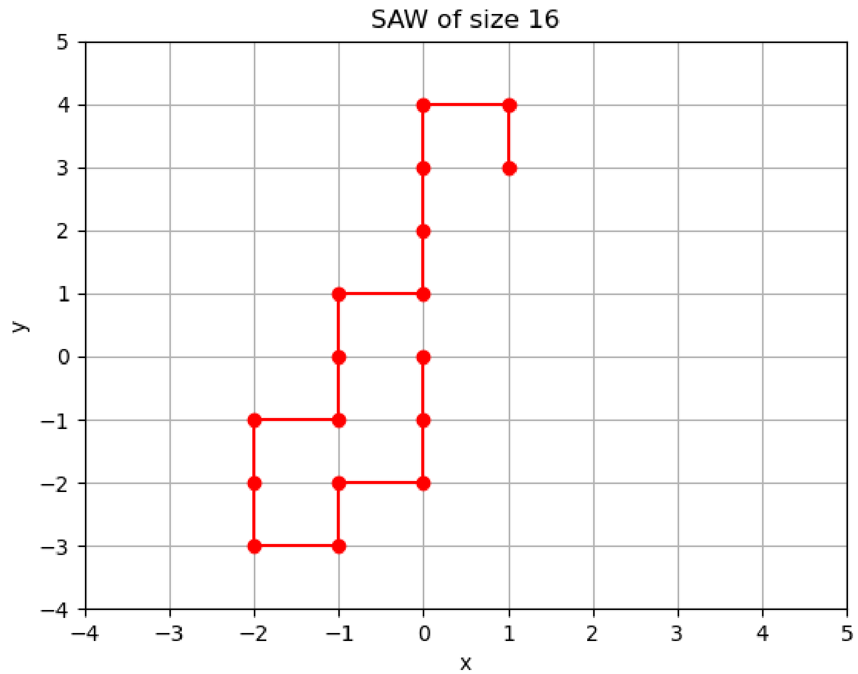

SAW of size 11
y
x

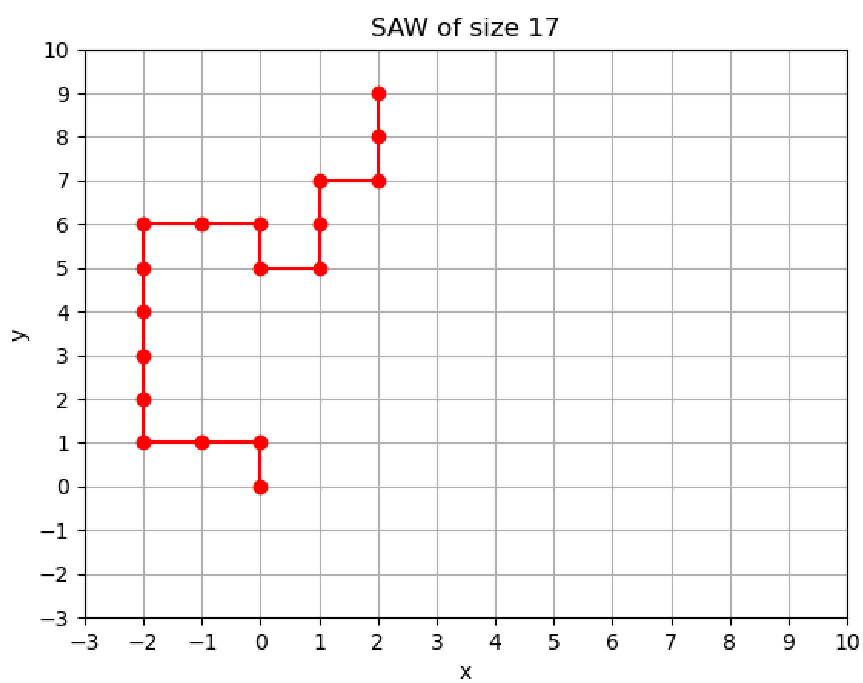

SAW of size 12
y
x
1
0
−1
−2
−3
−4
−5
−6
−7
−7
−6
−5
−4
−3
−2
−1
0
1

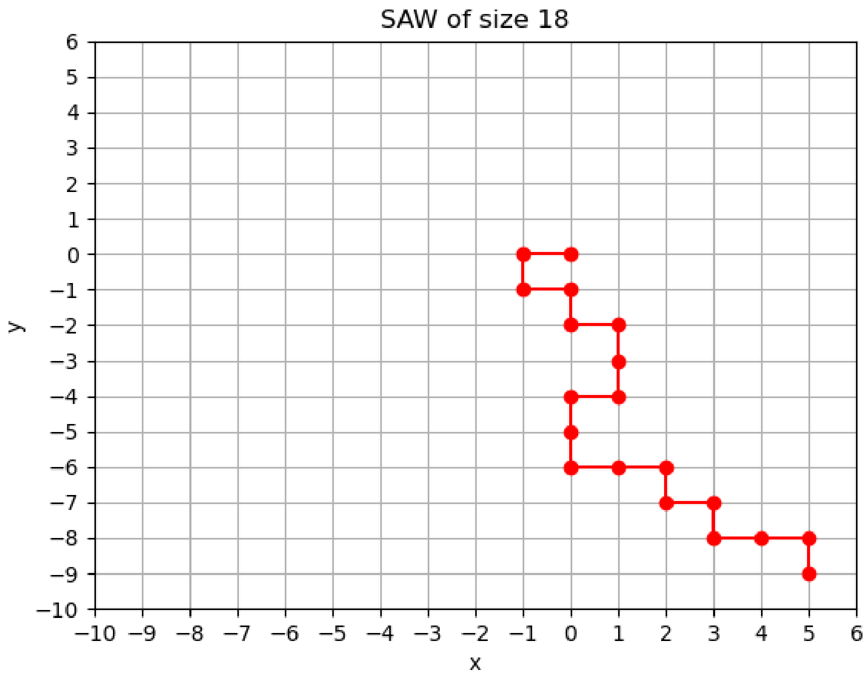

SAW of size 13
y
x
4
3
2
1
0
−1
−2
−3
−4
−5
−6
−7
−7
−6
−5
−4
−3
−2
−1
0
1
2
3
4

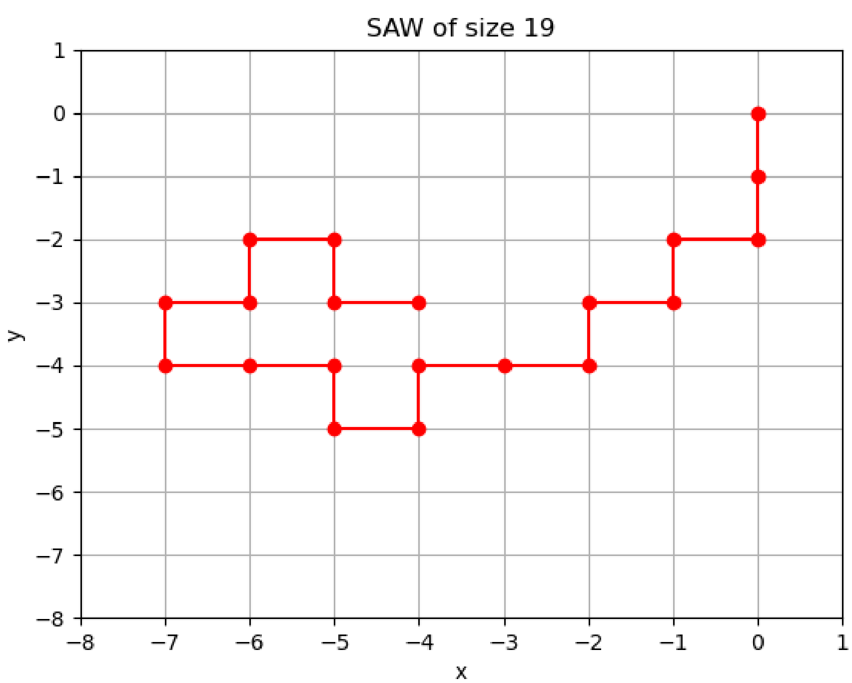

SAW of size 14
y
x
6
5
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4
5
6

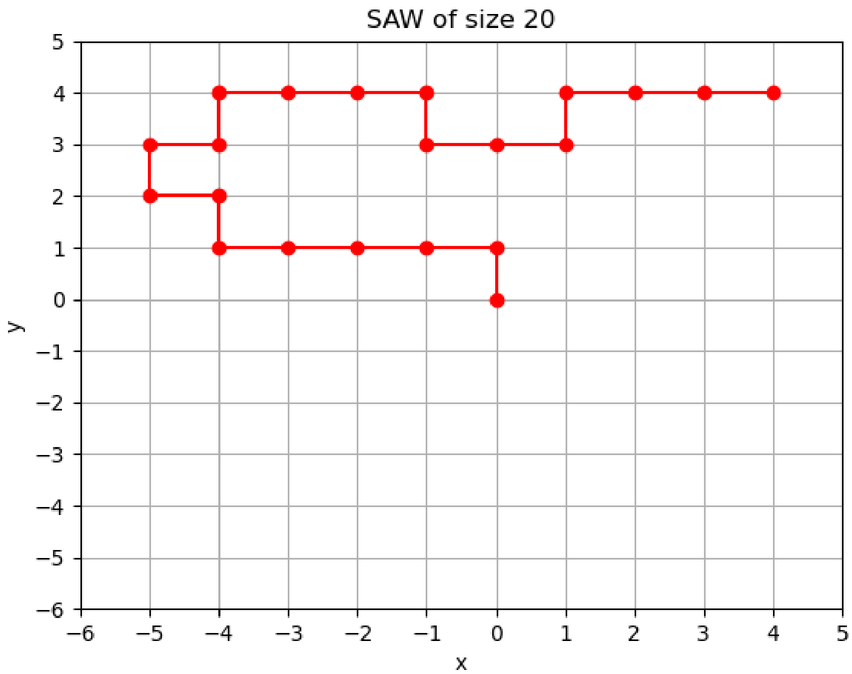

SAW of size 15
y
x
6
5
4
3
2
1
0
−1
−2
−3
−4
−5
−6
−6
−5
−4
−3
−2
−1
0
1
2
3
4
5
6


SAW of size 16
5
4
3
2
1
0
−1
−2
−3
−4
y
−4
−3
−2
−1
0
1
2
3
4
5
x


SAW of size 17
10
9
8
7
6
5
4
3
2
1
0
−1
−2
−3
y
−3
−2
−1
0
1
2
3
4
5
6
7
8
9
10
x


SAW of size 18
y
x


SAW of size 19
y
x

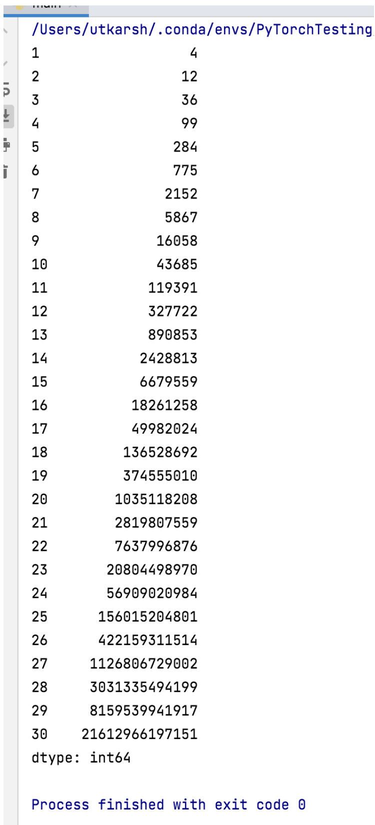


We can use this method to even compute the total count of the random walks Z_d for a given value of d. The following code snippet is doing the same:

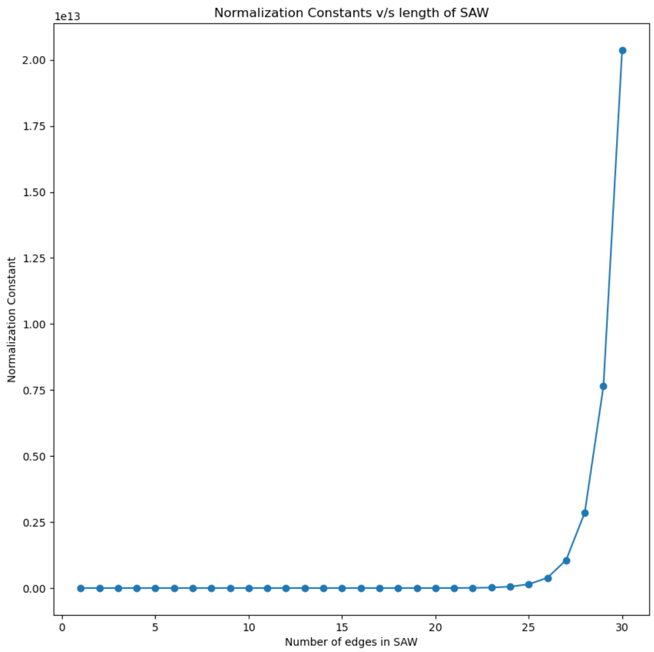
```
def GetNormalizationConstantWithoutResampling():
    wFrames = []

    n = 1000
    m = 5
    d = 30
    for i in range(n):
        status, x_frame, y_frame, w_frame = GetSelfAvoidingWalk(m, d, False)
        if (status == True):
            wFrames.append(w_frame)

    Ws = pd.concat(wFrames, ignore_index=True)

    dims = np.arange(1, d + 1)
    constants = Ws.mean()
    constants = (constants[1:]).astype(np.int64)

    print(constants)

    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
    ax.plot(dims, constants, marker='o')
    ax.set_title('Normalization Constants v/s length of SAW')
    ax.set_xlabel('Number of edges in SAW')
    ax.set_ylabel('Normalization Constant')

    grpahName = '/Users/utkarsh/NYU/Monte Carlo Methods/Homeworks/homework3/' +
                                                   'question2_normalization' + '.
                                                  png'
```

```
plt.savefig(grpahName)
```

Below is the image and graph with the count



```
/Users/utkarsh/.conda/envs/PyTorchTesting
1                4
2               12
3               36
4               99
5              284
6              775
7             2152
8             5867
9            16058
10           43685
11          119391
12          327722
13          890853
14         2428813
15         6679559
16        18261258
17        49982024
18       136528692
19       374555010
20      1035118208
21      2819807559
22      7637996876
23     20804498970
24     56909020984
25    156015204801
26    422159311514
27   1126806729002
28   3031335494199
29   8159539941917
30  21612966197151
dtype: int64

Process finished with exit code 0
```



We can clearly see that there is exponential relationship between length and count of SAW.