SHARIF UNIVERSITY OF TECHNOLOGY
PHYSICS DEPARTMENT

Problem Set 5

Student name: *Ali Fayyaz - 98100967*

Course: *Computational Physcis - (Spring 2023)*
Due date: *March 17, 2023*

Exercise 5.1

Prove equation 5.5 in the textbook.

Answer.

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2$$

$$\langle x(t) \rangle = \frac{l}{\tau}(p-q)t = Nl(p-q)$$

$$\begin{aligned}\langle x^2(t) \rangle &= \Big\langle x^2(t-\tau) + a^2 l^2 + 2al\,(x(t-\tau)) \Big\rangle \\ &= \langle x^2(t-\tau) \rangle + l^2 \langle a^2 \rangle + 2l \langle ax(t-\tau) \rangle\end{aligned}$$

$$\langle a^2 \rangle = p(+1)^2 + q(-1)^2 = p + q = 1 \tag{1}$$

$$\langle ax(t-\tau) \rangle = \langle px(t-\tau) - qx(t-\tau) \rangle = (p-q)\langle x(t-\tau) \rangle = \frac{l}{\tau}(t-\tau)(p-q)^2 \tag{2}$$

$$\begin{aligned}&\xrightarrow{(1),(2)} \langle x^2(t) \rangle = \langle x^2(t-\tau) \rangle + l^2 + \frac{2l^2}{\tau}(p-q)^2(t-\tau) \\ &\Longrightarrow \langle x^2(t) \rangle = \langle x^2(t-2\tau) \rangle + 2l^2 + \frac{2l^2}{\tau}(p-q)^2\left((t-\tau) + (t-2\tau)\right) \\ &\Longrightarrow \langle x^2(t) \rangle = \langle x^2(t-3\tau) \rangle + 3l^2 + \frac{2l^2}{\tau}(p-q)^2\left((t-\tau) + (t-2\tau) + (t-3\tau)\right)\end{aligned}$$

$$\xrightarrow{deduction} \langle x^2(t) \rangle = \underbrace{\langle x^2(t-N\tau) \rangle}_{\langle x^2(0) \rangle = 0} + \sum_{i=0}^{N-1} \left(l^2 + \frac{2l^2}{\tau}(p-q)^2(t-i\tau) \right)$$

$$= Nl^2 + l^2(p-q)^2(N-1)N$$

$$\Longrightarrow \sigma^2 = \underbrace{\left(1 - (p-q)^2\right)}_{(p+q)^2-(p-q)^2=(2p)(2q)} Nl^2 = 4pqNl^2 = 4\frac{l^2}{\tau}pqt$$

Exercise 5.2

Write a program for a 1 dimensional random walker and check equations 5.4 and 5.5 in the textbook for different values of p. Let one of these values be $p = 1/2$.

Answer. The equations were tested numerically. For each $p \in \{0.2, 0.4, 0.5, 0.6, 0.9\}$, 5000 1D random walkers were generated that each took 10,000 steps. Their positions were recorded 200 times (200 *snapshots* were taken), or equivalently every 50 steps, and the average position was calculated. Then $\langle x \rangle \sim (p - q)N$ and $\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2 \sim 4(pq)N$ were calculated and graphed for the 200 *snapshots* taken. To fit the data, *numpy.polynomial.polynomial.ploufit()* was used. The numerical analysis corresponded precisely with theory and the slope of the curves were as predicted by the aforementioned relations.

In order to generate the random walkers' position data, one can use many methods, two of which are as follows, the latter implemented:

1. **Explicit Boolean Indexing:**

   With N being the number of steps, p being the probability of taking a step in the positive x direction, the array *steps* is generated and the *position* is generated afterwards using the cumulative summation method:

   ```
   # Create an array with N elements, each with a random value in the half-open
   # interval [0,1)
   steps = numpy.random.random(N)
   # If the an element is smaller than p, replace it with -1, otherwise replace it with +1
   steps[steps < p] = -1
   steps[steps >= p] = 1
   # Add the steps together to get the position
   position = numpy.cumsum(steps)
   ```

2. **Implicit Boolean Indexing:**

   This utilizes the fact that *numpy.random.random()* creates an array with values in the interval [0,1), so by using a *mask* or *boolean indexing* one can turn the elements to boolean values:

   ```
   (np.random.random(N) < p)        # elements are either 0 or 1
   ```

   At this step, all the array elements are composed of booleans, *True* and *False*; or equivalently, *0*s and *1*s. Multiplying the array elements by 2, all the elements will be *0*s and *2*s:

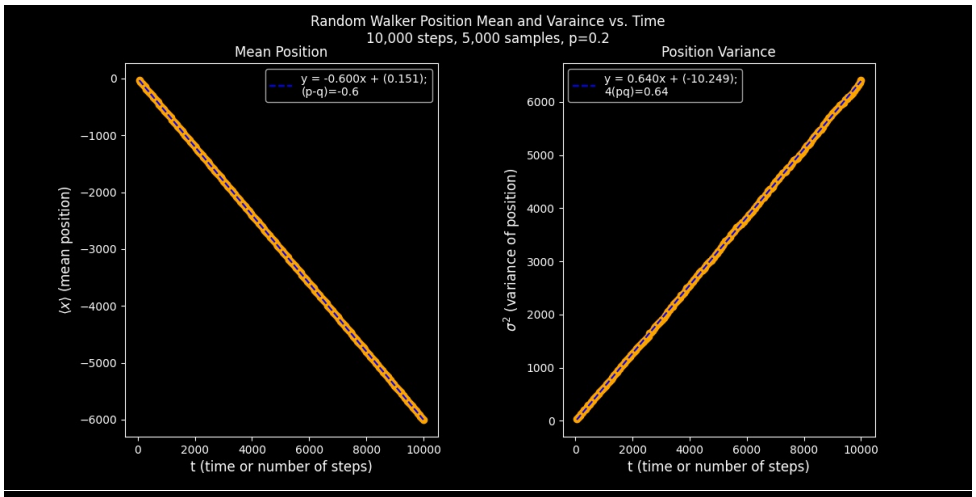
   ```
   2 * ( np.random.random(N) < p)        # elements are either 0 or 2
   ```

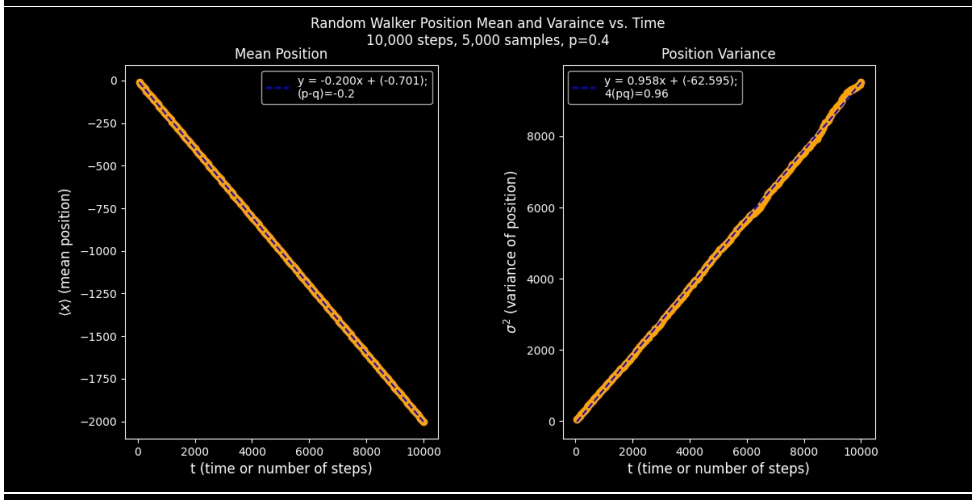
   Subtracting 1 will make all the elements either *-1* or *+1*.

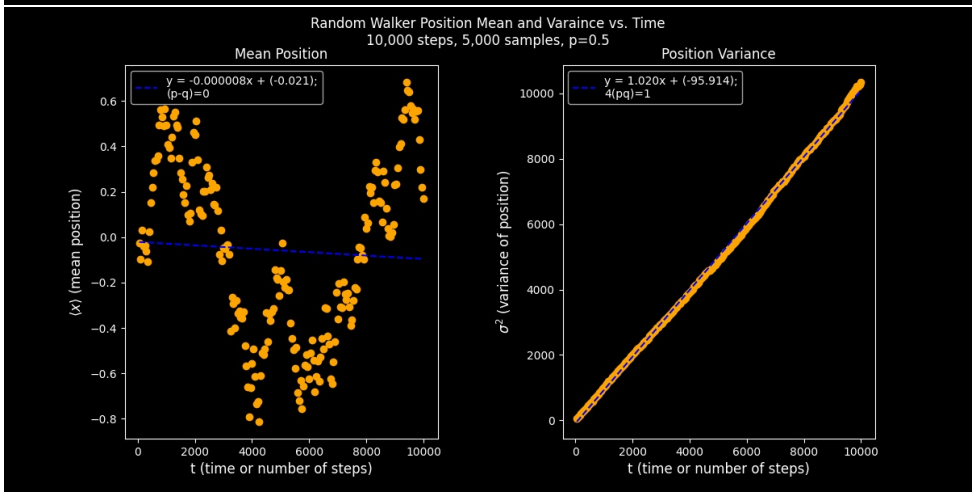
```
2 * (np.random.random(N) < p) - 1        # elements are either -1 or +1
```

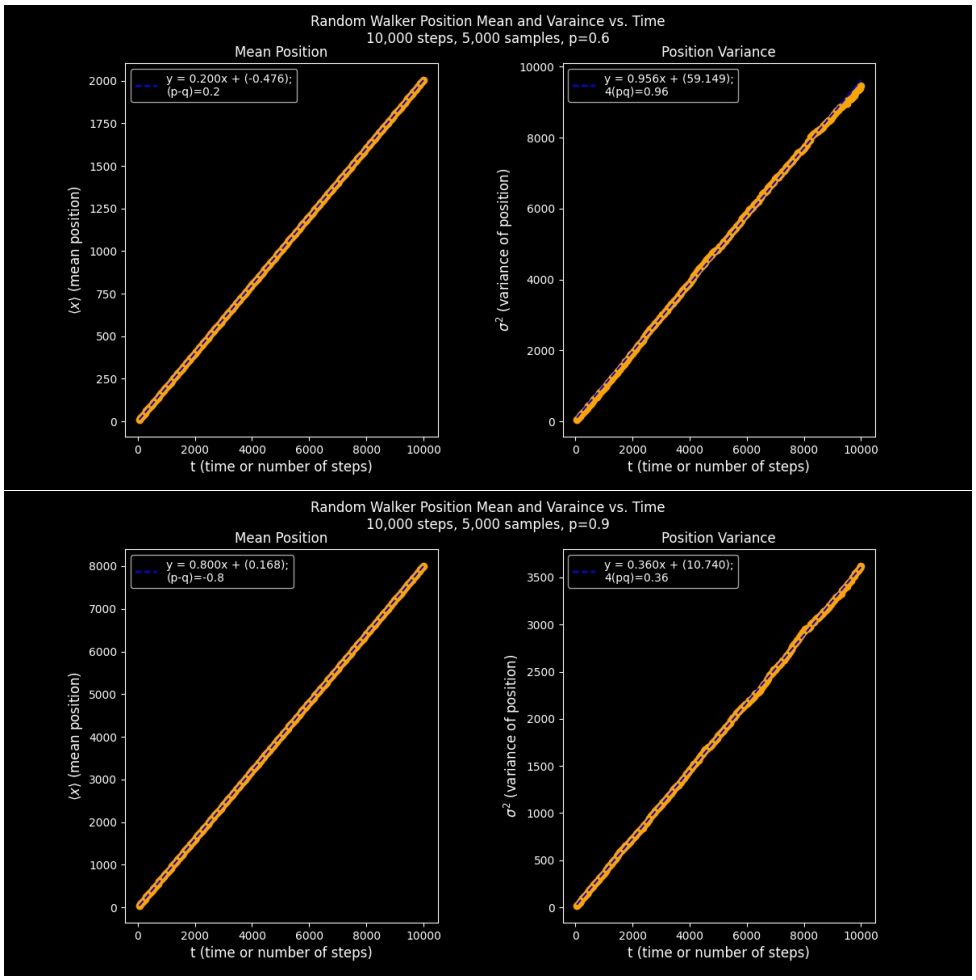
As the former method, adding the steps together will yield the position:

```
steps = 2 * np.random.random(N) < p
position = numpy.cumsum(steps)
```









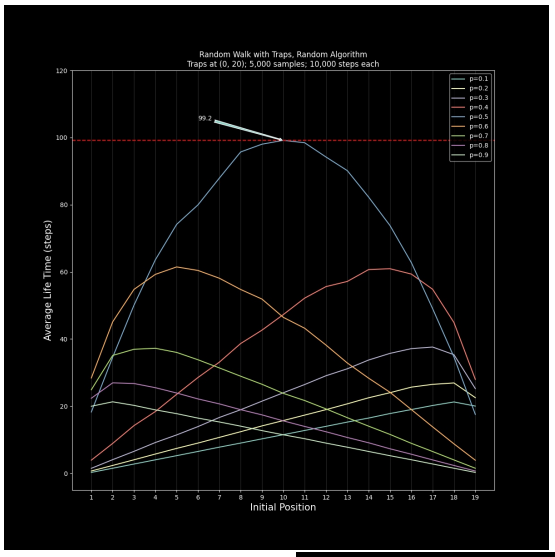
Exercise 5.3

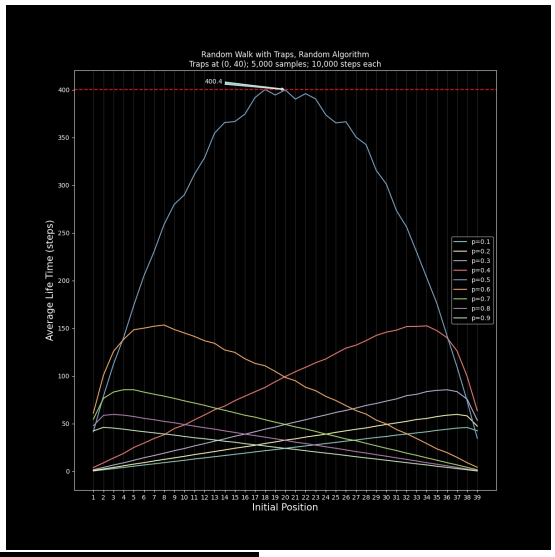
Put traps at the two ends of a 20-position grid. Calculate the average lifetime of a random walker for different values of p. Show that the average lifetime depends on the initial position of the walker on the grid.

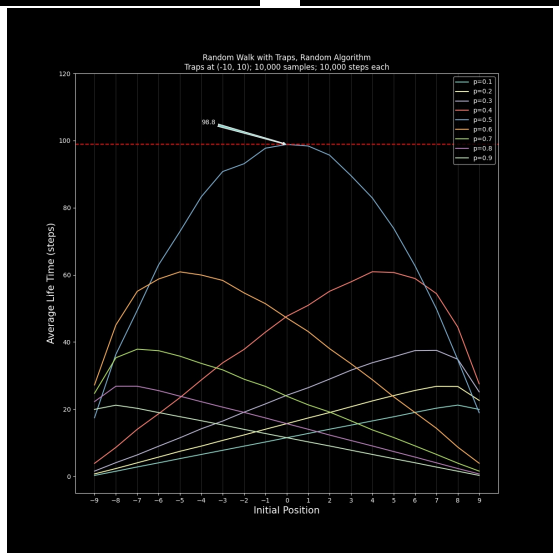
Answer. A function was defined to calculate the average lifetime of a number of random walkers. Obviously, to find a walker's lifetime is to count the number of steps it takes before reaching the position of the traps. For $p \in \{0.1, 0.2, ..., 0.9\}$ the data obtained from the function was graphed.

- **avg_life_time():**

  This function initiates a desired number of random walkers from all the available starting points within *traps* and returns their average life time. It takes a parameter p for the probability of a random walker to walk in the positive direction, a parameter N for the maximum number of random steps (It should be a large enough number so as to be quite sure that the walker will eventually step into the trap positions), another parameter *num_samples* to get the number of random walkers initiated, the average lifetime of which will be reported. Needless to say, the more samples, the more accurate the final results. The final parameter is *traps*, a tuple to determine the position of the traps on the coordinate; it can be set at desired positions and there is no need for it to be symmetric around 0.







Exercise 5.4

Use the census algorithm described in the textbook to answer the previous exercise.

Answer. The algorithm was implemented in the following function. This algorithm is deterministic so the need to average over different samples does not arise. This makes the algorithm much faster. The resulting graphs were similar to the previous exercise. Raising the *cut_off_probability* will give more accurate results.

- **rwt_census():**

  This function takes a similar parameter p as described earlier, another parameter *maneuver_length* as the length of the coordinate on which the walker can move. Another parameter *cut_off_probability* determines how many times the program should run; as long as the probability of finding the walker in the end points (the traps) is smaller then the cut-off probability, the program has to keep running. When it reaches the cut-off probability, it is safe to say the walker has slipped into the traps and died.

The function generates an array of length *maneuver_length* with all elements zero, except for one, the *initial position*, from which the walker starts, which has the value 1. In each iteration, the sum of the elements of the array is calculated, excluding the two elements on the two ends of the array, which are equivalent to traps and toghether hold the probability of death. The sum is then put into a variable which increases with every iteration. The mathematical reasoning behind this is as thus:

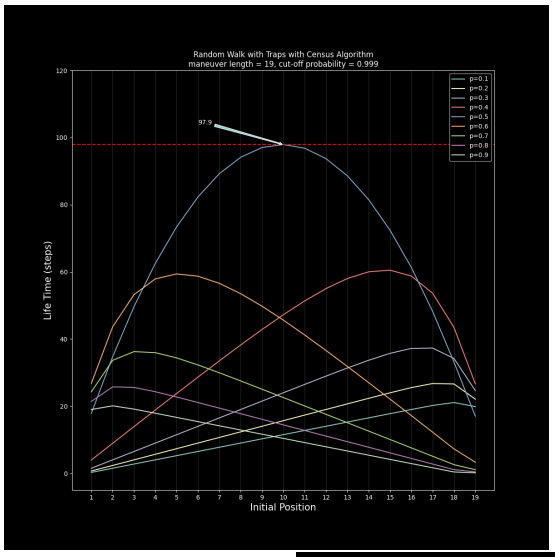
Let X be the lifetime of a random walker and $P(A_i)$ the probability of it being alive at the i-th step of its walk. Define the indicator variable I_i such that:

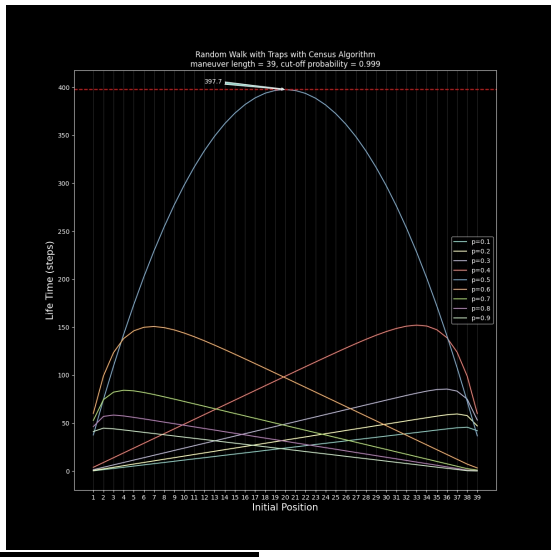
$$I_i = \begin{cases} 1, & \text{if the walker is alive and takes the i-th step} \\ 0, & \text{otherwise} \end{cases}$$

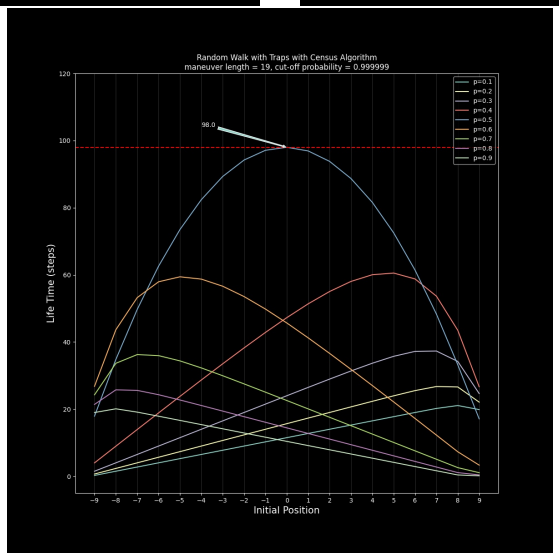
Then, the expected lifetime of the walker, or equivalently the number of steps it takes before dying, is:

$$E[X] = E\left[\sum_{i=1}^{max\_steps}(I_i)\right] = \sum_{i=1}^{max\_steps} E[I_i] = \sum_{i=1}^{max\_steps}((P(A_i)\times 1) + (P(A_i^c)\times 0))$$

$$E[X] = \sum_{i=1}^{max\_steps} P(A_i)$$





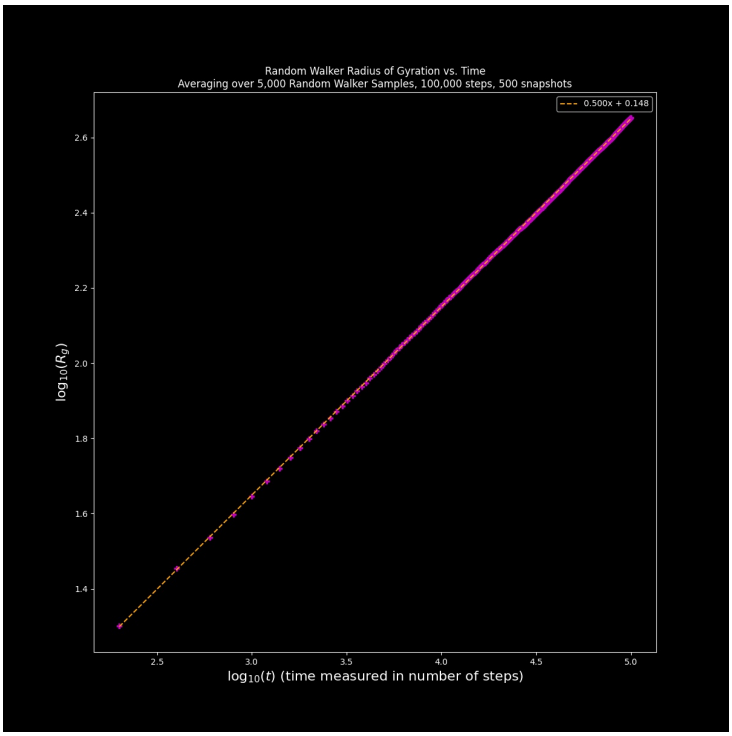


Exercise 5.5

Write a program for a 2D random walker on a square grid. Suppose taking a step in all directions is equally likely. Check equation 12 of the textbook with this walker.

Answer. In order to study the power law and obtain ν, 5,000 random walker samples were generated, each taking 100,000 steps. 500 *snapshots* were taken of each walker and the radius of gyration of the walkers was calculated at those moments, and then the average over all 5,000 samples was reported. As expected from theory, the radius of gyration grew with power $1/2$ of time:

$$R_g \sim t^{\nu} \qquad \Longrightarrow \qquad \log_{10}(R_g) \sim \nu \log_{10}(t)$$



Exercise 5.6

Write a program to show Diffusion-Limited Aggregation with a linear seed in 2 dimensions. Take the seed to be a line of length 200, and let the random walkers start walking from some height above the aggregation. Use color-coding and plot the aggregation.

Answer. The following function was defined to obtain a matrix which was given to *matplotlib.pyplot.imshow()* to plot the aggregation for different lengths and different number of samples. Moreover, out of curiosity, an aggregation model with a dot seed instead of a linear seed was also plotted using the function *dot_agg()*, which was basically a small modification, changing the boundary conditions on the former function.

- **diff_lim_agg():**

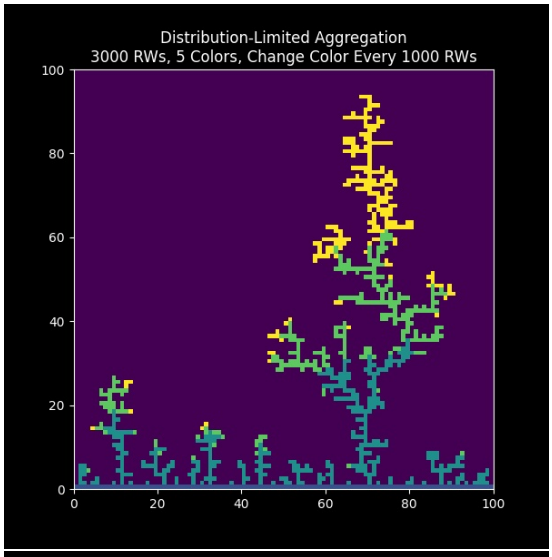
  This function takes the length of the linear seed, number of random walkers released into the matrix, aesthetic parameters like the number of colors that will be used in the image, and the number of random walk steps. It returns a matrix for the aggragation, with integer elements denoting colors, that will be used for plotting. The matrix is initiated with 1's in the bottom-most row for the linear seed, and 0's elsewhere.

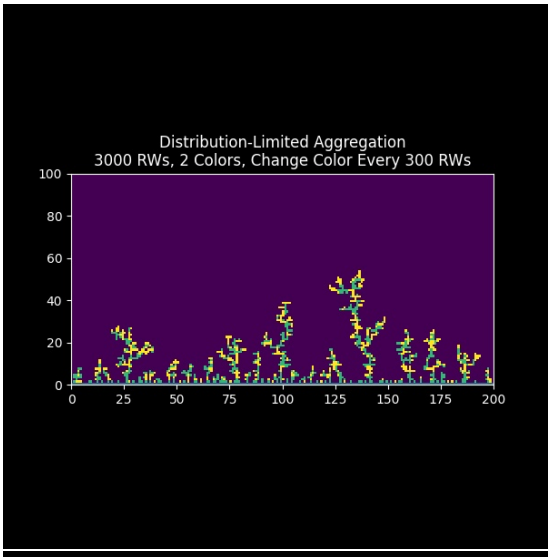
  The random walkers used take a step in the x direction and a step in the y direction in every *tick* of time. So if a walker is released from point (0,0), it can never do *rook walks* and go to points like (1,0), (0,1), (1,2), etc. This restriction on the walkers was imposed by the textbook, and to account for its effects (not allowing for the so-called*rook moves*), each random walker started its walk from a random x position throughout the matrix length. Also, each sample random walker is released from 10 positions above the highest branch of already aggregated particles.

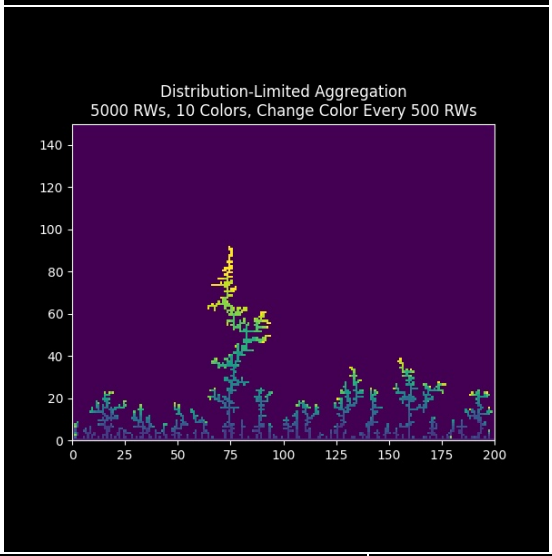
It is worth noting that if a walker happens to stray and get further away from a certain radius of the image, it is discarded and the next sample random walker is initiated.

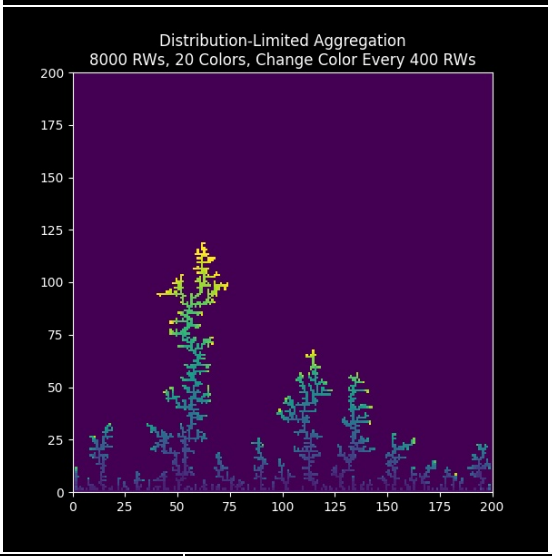
- **dot_agg():**

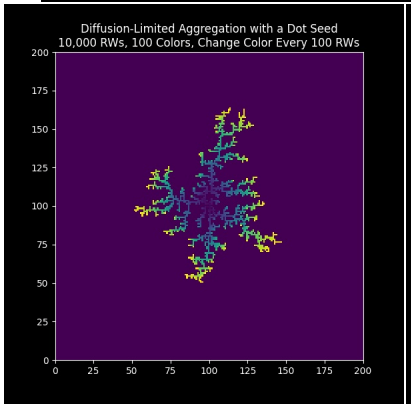
  This is a small modification of the function explained immediately above. It only differs in the boundary conditions. It starts with a dot seed in the middle of the image, one of the 4 sides of the image is randomly chosen to release the walkers, and are released from a distance of 10 of the aggregation.

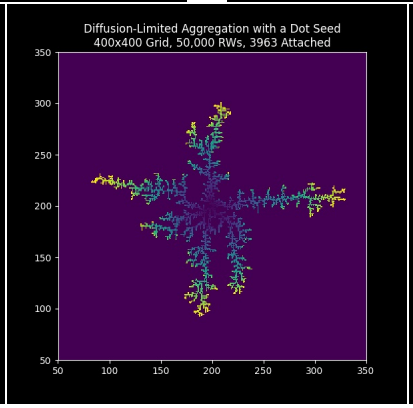


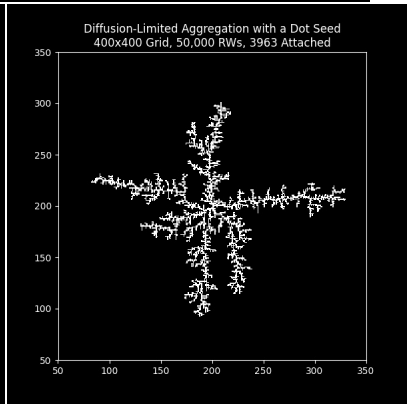












Exercise 5.7

Find all the self-avoiding random walks of length N. Plot the number of paths and compare with simple random walks.

Answer. The following recursive function was defined but it didn't work as expected. Due to tight deadlines, there was not enough time to debug it.

- **recursive_SAW():**

  The function takes the integer n as the number of the random walker steps, a matrix of booleans that denotes the network the walker walks on, and a final parameter *position*, denoting the current position of the random walker on the network.

  The network matrix is initiated with all elements *False*, except for the current position of the walker, which is *True*. With every given network and position, the function will check the four neighbors of that position to be available. It will recursively call itself and count the number of available positions and return it.