The Relation Between One and Two-Dimensional Random Walks to Pascal's Triangle and Analogous Higher-Dimensional Structures

Habiba Sorour, Maadi STEM Highschool for Girls

Abstract

This study aims to explore the use of Pascal's triangle and other analogous structures in higher dimensions to predict the position of particles inside a random walk after a certain number of steps. Pascal's triangle is a triangular array of numbers in which each number is the sum of the two numbers directly above it, and explores higher dimensional structures like Pascal's pyramid (the three-dimensional equivalent of Pascal's triangle). The first part of the study delves into the concept of Pascal's triangle and its fundamental properties. After that, Pascal's triangle was used in predicting the position of particles inside a random walk. A random walk is a mathematical model used to describe the movement of particles in a system where the direction and magnitude of each step are random. The probability of a particle ending up at a particular point after a certain number of steps in a random walk can be calculated using Pascal's triangle. The numerator of the fraction representing the probability is the value at the point in Pascal's triangle, while the denominator is the sum of the numbers in the corresponding row or layer of the triangle. The paper demonstrates how these concepts can be used to predict the position of particles inside a random walk and provide a real-life example of their application in the simulation of Brownian motion.

I. Introduction

Pascal's triangle is a triangular array of numbers that provide the coefficients in the expansion of the binomial formula $(x+y)^n$ [1]. Numbers are arranged in rows such that:

$$a_{nr} = \frac{n!}{r!\,(n-r)!} = \binom{n}{r} \quad (1)$$

where $\binom{n}{r}$ is the binomial coefficient (1). It is named after the mathematician Blaise Pascal, as he studied the triangle and published the "Traite du Triangle Arithmetique" in his studies of probability theory in the seventeenth century. To construct the triangle, a 1 is first placed on top. Each subsequent row in the triangle is created by adding the two entries diagonally above it (2).

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \binom{n-1}{r} + \binom{n-1}{r-1}. \quad (2)$$

The method of addition can be visualized by forming an upside-down triangle to get the value of the next element in the next row, where the apex is the sum of the two numbers on the other two vertices as shown in figure 1. Algebra, combinatorics, and probability theory all make use of the patterns placed in it.

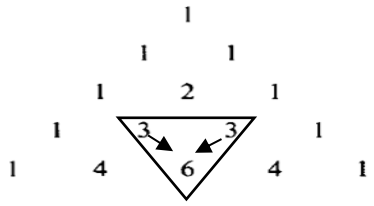


Figure 1: Pascal's triangle

Pascal's pyramid is the three-dimensional equivalent of Pascal's triangle and computes the trinomial expansion $(x+y+z)^n$. It is formed in a pyramid shape, where each subsequent layer is created by adding the three diagonals directly above as shown in figures 2,3 [2].

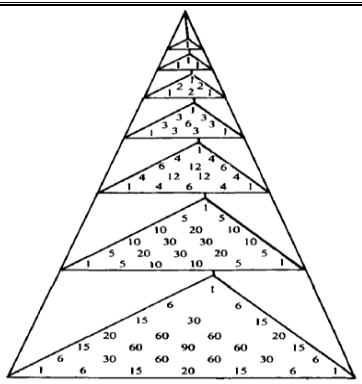


Figure 2: Pascal's pyramid in the form of layers [2]

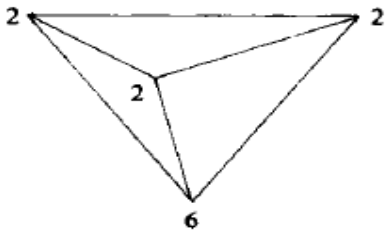


Figure 3: How each element is formed [2]

The purpose of this study is to see if Pascal's triangle and other analogous structures in higher dimensions can predict the position of particles inside a random walk after a certain number of steps. The relation between Pascal's pyramid and a two-dimensional random walk, which is determining the position of a point after a specified number of steps in two dimensions, is discussed to see if it is connected as Pascal's triangle and one-dimensional random walk and if it can be applied in higher dimensions and in real life. A real-life example is the Wiener process also called Brownian motion, which is the random movement of particles inside a fluid. It can be simulated by two-dimensional random walks, which include the path a molecule can take in its motion.

II. Theory

i. Patterns in Pascal's Triangle

Firstly, the triangle is symmetrical, where the numbers are placed like a mirror image. Furthermore, The sum of elements in a row equal 2^n. Moreover, the diagonals have patterns inside them. The first diagonal is just "1" s. The next diagonal is the counting numbers. The third diagonal is the triangular number which are fictitious numbers that can be represented as an equilateral triangular grid of elements with each row having one more element than the one before it.

Lastly, the fourth diagonal represents the tetrahedral numbers which are the number of balls needed to form a tetrahedron. If you add up the shallow diagonals, the Fibonacci sequence shows up.

The Fibonacci sequence is a sequence of numbers in which each number is the sum of the two numbers before it. Figure 4 shows all the patterns just mentioned.

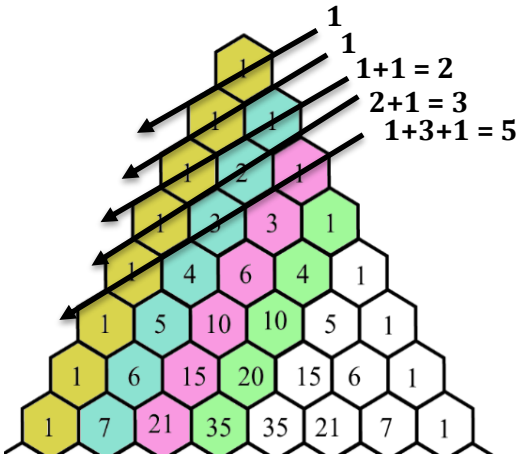
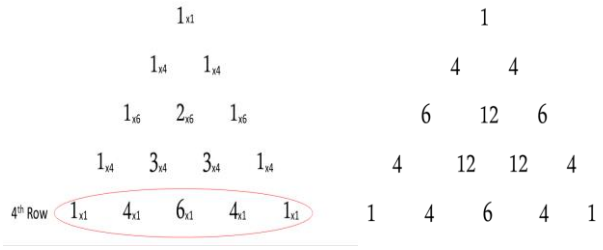


Figure 4: Patterns in Pascal's Triangle

ii. Patterns in Pascal's Pyramid

To visualize the numbers in the pyramid, they are divided into layers instead of rows, which are present in Pascal's triangle. The sum of elements in layer n of Pascal's pyramid is 3^n. The numbers along the faces of the pyramid in the n^{th} layer of the pyramid are the same as Pascal's triangle elements in the n^{th}row. Another tie



(a): Pascal's triangle *(b): Layer 4 of Pyramid*

Figure 5: The relation between Pascal's pyramid and Pascal's triangle

between Pascal's pyramid and Pascal's triangle is that to obtain the elements in the n^{th} layer inside of Pascal's pyramid, constructing Pascal's triangle to the n^{th} row, then multiply each row by the numbers in the n^{th}row.

Figures (5a) and (5b) show an example of this relationship [3].

iii. Randomness and Random Walks

A random walk is the process of determining the position of a point after a specified number of random steps. The walk can be in one dimension, two dimensions, or three dimensions and higher dimensions. It is central to statistical physics and an important part of probability theory in mathematics.

iv. Random Walk in One Dimension:

A random walk in one dimension can be constructed as follows: a particle is present on the number line, and it has the probability of either going up or down. This can be visualized by flipping a coin; if it lands on head, the particle goes up, and if it lands on tails, the particle goes down [4]. There is an equally probable chance of landing on either heads or tails in both cases. Let s be the direction the point moves to (either up or down) and "d" the distance traveled by the point. "d" can be positive or negative, as the point can go above or below zero, and n is the total number of steps taken. (3).

$$d = s_1 + s_2 + s_3 + \ldots + s_n. \quad (3)$$

Random walks in one dimension have a relationship between Pascal's triangle as the probability of ending at a point equals each element in Pascal's triangle divided by the sum of the values in the row. Letting P(N) be the probability function, For instance, after 2 steps,

$$P(2) = \frac{1}{4}$$
$$P(0) = \frac{2}{4} \quad (4)$$
$$P(-2) = \frac{1}{4}$$

The numerator of the probabilities corresponds to row 2 in pascal's triangle and the denominator is the sum which equals 2^2 which is 4. Figure 6 shows an example of a random walk with 100 steps. To get the probability, row 100 of the triangle would be computed, the denominator would be computed by 2^{100}, and the nominator would be the elements in the row.

v. Random Walk on Two Dimensions

Random walks in two dimensions are simulated on two coordinates, x and y. The walk starts at the origin and takes N steps in the xy plane (in unit length, not in coordinate form). The radial distance R from the starting

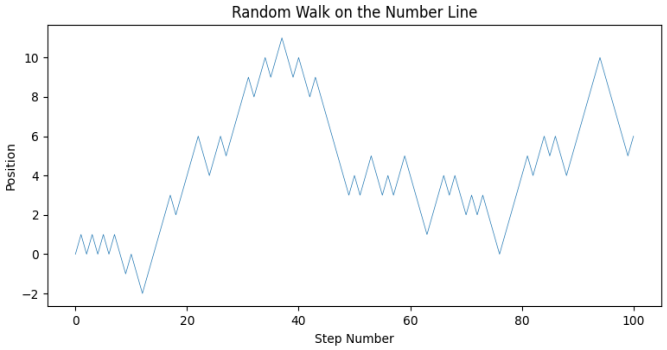


Figure 6: Random walk in one dimension

point after N steps (5):

$$R^2 = (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2 \quad (5)$$

Because the particle is in random motion, each movement position is equally likely to occur. The average radial distance after many steps will be where the root mean square of the step size (r_{rms}) [5] is (6):

$$R_{rms} \approx \sqrt{N} r_{rms} \quad (6)$$

And after a huge number of steps, the point is more likely to be at the origin position. Two dimensions random walks are present in our daily life as the diffusion of particles in space. Figure 7 shows a (8)
random walk after 100 steps.

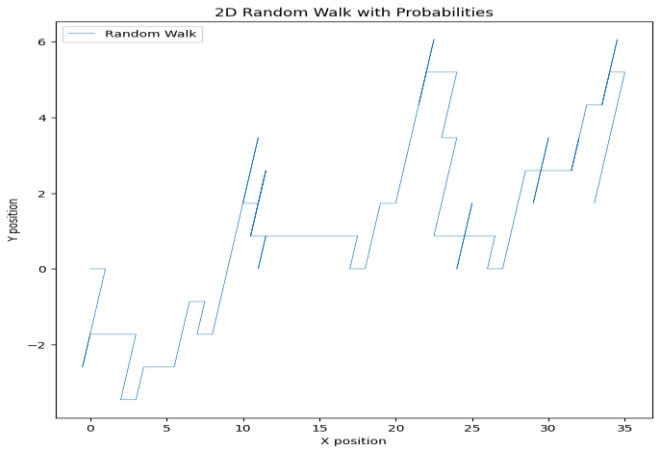


Figure 7: An example of a 2D random walk

vi. Real World Applications of Random Walks (Brownian Motion)

Brownian Motion is the random movement of particles inside a fluid. It gets its name from the Scottish botanist Robert Brown. He observed pollen grains under the microscope and said that the particles displayed rapid and irregular movement that seemed to be continuous. Albert Einstein and Marian Smoluchowski further developed Brownian Motion. Einstein was the first to realize that due to collisions with molecules, particles should move with kinetic energy equal to $k_B T / 2$ (where k_B is the Bolzmann constant and T is the fluid temperature).

Smoluchowski, on the other hand, used kinetic theory to perform calculations. He stated that one collision alone would not be sufficient to generate movement, but the number of collisions is large and can create motion. Smoluchowski realized that there will always be unbalance due to fluctuations in the order of the square root of the number of collisions, thus creating motion.

The other part of Einstein's theory was a result of Stokes law for the frictional force exerted on a body moving in a liquid: a spherical body of radius *a* moving with velocity *v* in a fluid of viscosity η is slowed down by a force given by $-6\pi\eta a v$.

Einstein formulated that the particle performs irregular motion, very similar to a random walk. It is characterized by the square displacement [$\Delta x(t) = |x(t) - x(0)|^2$] (where $x(t)$ is the position of particle at time *t)]* of the particle as it grows on linearly in time [6]:

$$\langle |\Delta x(t)| \rangle \approx 6Dt \qquad (7)$$

Equation (2.4a) is Einstein- Smoluchowski law of diffusion, and to derive the diffusion coefficient *D*, using Stokes law and energy equipartition equation (8)

$$D = \frac{k_B T}{6\pi\eta a v} \qquad (8)$$

To connect the ideas of a random walk and Brownian motion, the Wiener process is used. Although this is strictly speaking a confusion of a model with the reality being modeled, the Wiener process is occasionally referred to as "Brownian motion". The Wiener process is the scaling limit of a random walk; if there is a random walk with a very small step size, there a Wiener process can be approximated. If the step size is ε, for example, to approximate a Wiener length of L, a walk of length L/ε^2 is taken. The random walk turns into a Wiener process as the number of steps rises. This step is made to simulate the random movement of particles as the step size gets decreased to a small step size. It is controlled by the central limit theorem. Central limit theorem is that the sampling distribution of the mean will always be normally distributed, if the sample size is large enough.

Brownian motion can be observed in multiple parts of daily life. Gas molecules represent Brownian motion. Moreover, the motion of pollen grains in still water.

III. Methodology

i. Libraries

Built-in libraries of Python were used to facilitate the writing process of building and constructing the algorithms. To begin, the *math* library was used to implement mathematical values like *math.pi and math.cos()* were used. Secondly, the *random* library was used to implement the random factors in the random walk like *random.seed() and random.choice()* were used. "*random.choice"* is a built-in random module that chooses between the given arguments. The *matplotlib.pyplot* was used as a way to visualize the results in graph form in one dimension and in two dimensions. Moreover, the *numpy* library was used to add more mathematical functions like *sqrt().*

ii. Pascal's Triangle Algorithm

A function named "*pascal triangle*" was built that takes an argument "*n*" generating the nth row of Pascal's triangle. First, it initializes the current row to be (1), the first row of Pascal's triangle. Then, for each subsequent row, it creates a new list "*next_row*" and sets the first element to (1).

Figure 8 shows the key snippet that codes for Pascal's triangle. The first line creates a loop that iterates over each index "I" in '*current_row'* list, except the last index. "*len(current_row)* – 1" gives out the number of

elements inside the list minus the last element, as the last element is one. The term inside the for loop is the main part of the code, and it appends the sum of the current element "*current_row[i]"* and the next element "*current_row [i + 1]" to the next_row list.* Moreover, it sets *'current_row'* to *'next_row'* to generate the following row in the triangle. Lastly, it repeats until it reaches the n^{th} row of the triangle.

```
for i in range(len(current_row) - 1):
      next_row.append(current_row[i] +
current_row[i + 1])
```

Figure 8: Getting the values of Pascal's triangle

To code for the probability inside the triangle, a dictionary was initialized to store the values of the probabilities. They were computed by taking each value in the "*current_row"* list and dividing it by 2^n. Then, each probability gets added to the *"random_dict"* dictionary as shown in figure 9.

```
 random_dict = {}
    for value in current_row:
      prob_2d = value / 2**n
      random_dict[value] = [prob_2d]
      random_dict[value, y] =
random_dict.pop(value)
      y -= 2
```

Figure 9: getting the probabilities in pascal's triangle

Tying the random walk placement to Pascal's triangle was done by adding a key value in the dictionary that contained the probability values. This was done by making a new variable called *y* that starts by having the same value as the number of steps taken, after that it is decremented by a value of two, then it creates a new key-value pair were each probability has two keys, one being the value inside pascal's triangle and the other key is the position after a set number of steps. For instance, after 4 steps, the *'random_dict"* dictionary has values of "*{(1, 4): [0.0625], (4, 2): [0.25], (6, 0): [0.375], (4, -2): [0.25], (1, -4): [0.0625]}"* where the first element in the tuple is the value in the pascal's triangle in pascal's triangle, and the second element in the tuple is the possible End Points After *N* Steps.

iii. Random Walk In One Dimension Algorithm

A function named "*random_walk_1d"* was constructed to simulate the random walk. It takes two arguments: "*start"* which initializes the position of the walk, and "*n*", which is the length of the step. Firstly, *random.seed()* was used to ensure that each time the function gets called, the number generated is different. The "*current"* variable gets assigned the "*start"* variable to initialize the starting position. After that, a list "*steps"* gets initialized that stores the steps taken by the program.

Figure 10 shows the loop that iterates "*n*" times, generating a random step and updating the position of the random walk. "*random.choice([-1, 1])"* is the two possible outcomes of the walk, either going to the left or to the right. After the choice is made, the current position gets updated by adding the random step to the current position and appending it to the steps list.

```
for i in range(n):
        step = random.choice([-1, 1])
        current += step
        steps.append(current)
```

Figure 10: Random walk in one-dimension code

To calculate the probabilities of the random walk, '*num_trials'* is specified, which signifies the number of times the random walk would be executed. The starting position '*start'* gets assigned the value zero to indicate starting at the origin. After that, it records the final position of the particle after each trial and keeps track of the frequency of each final position in a dictionary called '*end_points_freq_dict'.*

A for loop gets initiated that runs the walk a specified number of times, (set to 1000 as a baseline) and records the final position of the particle after each trial inside the *end_points_freq_dict* dictionary. To calculate

the probability of each final position, the frequency of each position is divided by the *'total_trials'.* The resulting probability is then stored in a new dictionary called *'end_points_prob_dict"as shown in f*igure 11.

```
start = 0

num_steps = int(input("Enter the number
of steps: "))

num_trials = 10000

end_points_freq_dict = {}

for i in range(num_trials):

    steps = random_walk_1d(start,
num_steps)

    end_point = steps[-1]

    end_points_freq_dict[end_point] =
end_points_freq_dict.get(end_point, 0)
+ 1



total_trials = num_trials

end_points_prob_dict = {}

for point in end_points_freq_dict:

    end_points_prob_dict[point] =
end_points_freq_dict[point] /
total_trials
```

Figure 11: Calculating the probabilities inside the random walk

To add the uncertainty (standard error) to the graph, using the formula

$$\sqrt{\rho \times (1-p)/n} \qquad (9)$$

Where *p* is the probability of ending at a point, *(1-p)* is the probability of the complementary event, and *n* is the total number of trials. The standard error is a measurement of the deviation between the sample mean and the actual population mean. This was calculated to see the range of error that the probability gets to measure. The probability values are first extracted by the code using the "*values ()"* method from a dictionary called "*end_points_prob_dict,*" and then they are transformed into a list using the "*list ()"* function. The variable "probs" holds the resulting list of probabilities.

The standard error is then calculated for each probability value in the probs list using a list comprehension as shown in figure 12. The square root of each probability value p is calculated using the *sqrt ()* function from the NumPy library. The *list 'stderrs'* contains the resulting standard errors.

```
probs =
list(end_points_prob_dict.values())

stderrs = [np.sqrt(p * (1 - p) /
total_trials) for p in probs]
```

Figure 12: Standard error in the random walk

iv. Pascal's Pyramid Algorithm

The code for Pascal's pyramid was taken from *bodyshots on Git Hub* [7] with changes to fit the model. The code has two classes. The first class "*Pascal3DElem"* codes for the element inside Pascal's pyramid, which is a number. The second class "*PascalPyramid"* codes for Pascal's pyramid itself. "*PascalPyramid"* has three methods, each one codes for a part to generate the pyramid.

The first method "*__init__"* initializes the layer's attribute of the pyramid, which is a list of *"Pascal3DElem"* objects representing the pyramid. It takes an argument that signifies the layer to generate the layers included in the pyramid. It uses a loop to generate each layer of the pyramid, starting from layer "0", which is (1). After that, the loop calls the *get_next_layer* method to generate each subsequent layer of the pyramid.

The second method, *"get_next_layer,"* generates the next layer of the pyramid by being given the previous layer; it takes *'prev_layer',* which represents the previous layer of the pyramid. The method creates an empty list called *'next_layer'* to contain the next layer of the pyramid. Moreover, a loop is used to generate each row of the next layer, starting with the top row, creating a list of *Pascal3DElem* objects for

each row, with the length of the list equal to the row number plus one. Furthermore, the method uses another loop to calculate the value of each element in the next layer based on the values of the elements in the previous layer. For each element in the previous layer, the method adds the value to the values of the elements that are above, below, and right next to it in the next layer, generating the next layer of the pyramid. Figure 13 shows the method in detail.

```
def get_next_layer(self, prev_layer):

        next_layer = []

        for i in
range(len(prev_layer[0]) + 1, 0, -1):

            next_layer.append([Pascal3D
Elem() for _ in range(i)])


        for row_num in
range(len(prev_layer)):

            for entry_index in
range(len(prev_layer[row_num])):

                curr_elem =
prev_layer[row_num][entry_index].elem

                next_layer[row_num][ent
ry_index].elem += curr_elem

                next_layer[row_num][ent
ry_index + 1].elem += curr_elem

                next_layer[row_num +
1][entry_index].elem += curr_elem

        return next_layer
```

Figure 13: Generating the next layer of pascal's pyramid

The last method, "*probability_pyramid,*" is used to create the probability of ending at a point inside the pyramid. It takes the last layer of the pyramid, then generates a list of values of all elements in the last layer. It creates an empty dictionary to store the values. Moreover, it uses a loop to iterate over the list of elements, and for each element, it divides the element by 3^n, where the n represents the number of layers in the pyramid. It takes this value and add it to the "*prop_3d*" dictionary with the key being the original value and the value is the probability as shown in figure 14.

```
def probability_pyramid(self):

      last_layer = self.layers[-1]

      elements = [elem.elem for row in
last_layer for elem in row]

      global prob_3d

      prob_3d = {}

      y=1

      for elem in elements:

        divide_3d= elem / (3**x)

        prob_3d[elem] = [divide_3d]

        y += 1

      return elements , prob_3d
```

Figure 14: Getting the probability in Pascal's pyramid

v. Random Walk in Two Dimensions

A *"random_walk_2d"* function was created to generate a random walk starting from the origin and having to move between three possible steps. The steps were designed to align with the values of the pyramid, as the pyramid has three faces.

The function takes two arguments, '*start*', a tuple to indicate the starting position, and '*num_steps*', an integer specifying the number of steps to take. A loop is initiated that takes '*num_steps',* where in each step, the function generates a random step size. The direction gets assigned by calling *'random.choice()',* where it contains three tuples representing the direction of the step. The steps can have three directions, and the angle between each step is 120 degrees. This was made by using trigonometry and making the directions *(sin 30, cos 30), (1,0), and (-sin 30, -cos 30).* Then, the function updates the current position by adding the step size and direction to the x and y coordinates in position. They were then

separated to ease graphing of the position as shown in figure 15.

```
for i in range(num_steps):

        step =
random.choice([(0.5,math.cos(math.pi /
6)) ,(1,0),(-0.5,-math.cos(math.pi /
6))])

        position[0] += step[0]

        position[1] += step[1]

        steps.append(tuple(position))

        x_positions.append(position[0])

        y_positions.append(position[1])
```

Figure 15: Random walk in two dimensions

Similar to how the probabilities were calculated in the one-dimensional random walk, the probability of the two-dimensional random walk is constructed. The '*start'*, instead of being an integer, is now a list consisting of two zeros to indicate the origin point. Since the directions set have decimal points, the final position of the particle is rounded to 5 decimal places and stored as a tuple in the "*end_point"* variable before being added to the "*end_points_freq_dict"* dictionary. This makes sure that there are no random errors or duplicates at the same point as shown in figure 16.

```
start = [0, 0]

num_steps = int(input("Enter the number
of steps: "))

num_trials = 1000

end_points_freq_dict = {}

for i in range(num_trials):

    steps, x_positions, y_positions =
random_walk_2d(start, num_steps)

    end_point = tuple(round(coord, 5)
for coord in steps[-1])

    end_points_freq_dict[end_point] =
end_points_freq_dict.get(end_point, 0)
+ 1



total_trials = num_trials

end_points_prob_dict = {}

for point in end_points_freq_dict:

    end_points_prob_dict[point] =
end_points_freq_dict[point] /
total_trials
```

Figure 16: Calculating the probability in two dimensions

vi. Testing the Models

After building the models, they were subjected to visualization to see the relation between random walks in one and two dimensions and Pascal's triangle and Pascal's pyramid, to see if it can predict the probability of ending at a point in the random walk based on the values inside of the structure.

The random walk was simulated a thousand times, and then ten thousand times, in both the one-dimensional random walk and the two-dimensional random walk. The standard error ($\hat{p}$) of the probabilities given by the random walk was given by the equation (10) where p is the frequency of ending at the point and n is the total trials given.

$$\hat{p} = \sqrt{p(1-p)\,/\,n} \qquad (10)$$

For the two-dimensional random walk, error bars were added to the graph representing the uncertainty in the estimated of the probabilities of ending at each point. They indicate the standard deviation of the estimated probabilities, which is a measure of how spread out the estimates are around their mean, that's why the standard deviation and mean was calculated. The standard deviation was calculated by Bernoulli distribution, which aligns with equation (3.6). This formula assumes that the probability estimates are independent and identically distributed, which is a

reasonable assumption in this case since each trial is independent and the endpoint probabilities are estimated using the same number of trials figure 17.

```
x_means = np.array([point[0] for point
in end_points_prob_dict.keys()])

y_means = np.array([point[1] for point
in end_points_prob_dict.keys()])

x_stdevs = np.array([np.sqrt(prob * (1
- prob) / total_trials) for prob in
end_points_prob_dict.values()])
```

Figure 17: Standard Deviation calculated in the graph

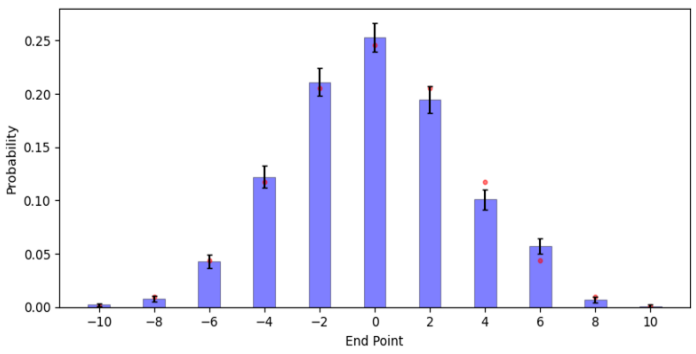
IV. Results and Discussion

i. Pascal's triangle and one-dimensional random walk

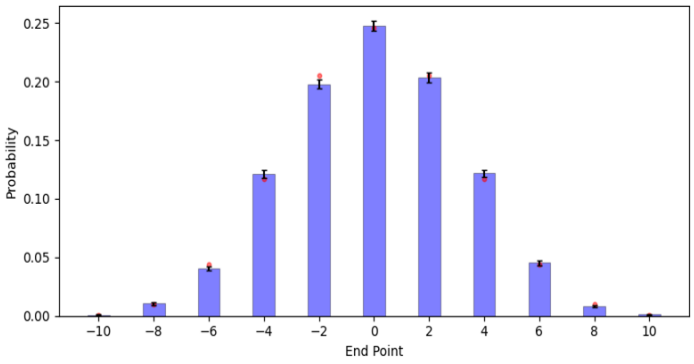
There is a set relation between Pascal's triangle and a one-dimensional random walk where the probability of ending at any point after *n* steps corresponds to the values in the triangle over their sum.

$$v = \frac{1}{2^n} \qquad (11)$$

The numbers were then compared to Pascal's triangle probability numbers *v* and it showed that by increasing the number of trials, the numbers approached the values given out by Pascal's triangle. This is apparent in Figure 18, where in (a) the probabilities don't align as well as in (b). This shows that Pascal's triangle shows the probability of landing at each point accurately. Looking at the graph, a pattern emerges where the smallest number is pascal's triangle, which is (one) corresponds to the farthest point from the origin.



(a). After a thousand trials



(b). After ten thousand trials

Figure 18: One-dimensional random walk after 10 steps

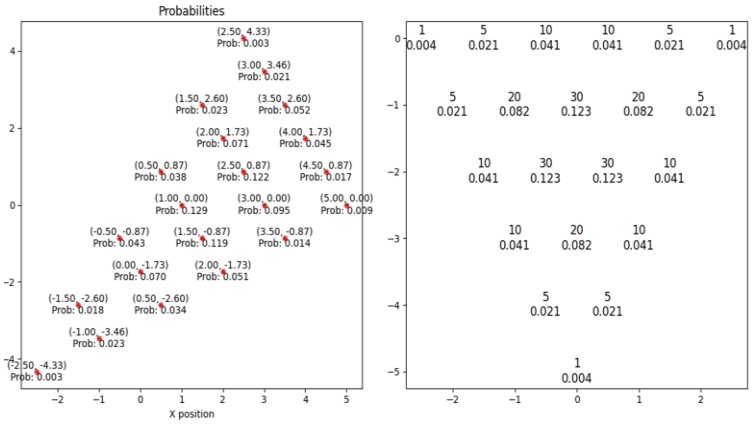
An observation can be made were the middle values inside of the triangle are nearest to the origin. Furthermore, in rows where there is an odd number of elements in the row, the middle element corresponds to the highest probability which is returning to the origin.

ii. Pascal's Pyramid and Two-Dimensional Random Walk

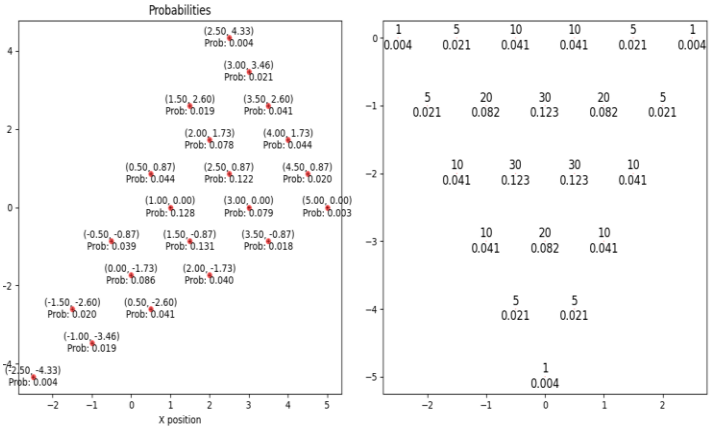
In two dimensions, similar observations can be made with Pascal's Triangle and the one-dimensional random walk. To begin with, the probabilities of ending at any point in the walk showed like the values given out by the pyramid given by the formula, where v is the probabilities given out by the pyramid.

$$v = \frac{1}{3^n} \qquad (12)$$

Like the one-dimensional walk, the probabilities whenever the number of trials increases, the probability of ending at points closely resembled those of pascal's pyramid.



(a) After a thousand trials



(b) After ten thousand

Figure 19: Two-dimensional random walk probability

The values in each row in the probabilities correspond to a row inside of the layer in pascal's pyramid. As it is clear in figure 19, the topmost points in the graph correspond to the top most row in pascal's pyramid layer. Each layer is computed where the layer number is the number of steps taken.

The closer to the origin, the higher the probability to end at it. The values in a layer inside of the pyramid has its higher values in the middle of the layer. This corresponds to the results given.

V. Conclusion

Pascal's triangle is one of the fascinating subjects taught in mathematics, it has usages in combinatorics and algebra. This paper proved that there is another way that pascal's triangle and other structures in higher dimensions like pascal's pyramid can be used to predict the probability of ending at a point inside random walks in one dimension and two dimensions. The purpose of this study was to see if pascal's triangle and higher dimensional structures like pascal's triangle can model high dimensional random walks. This relation was set after building the probability inside of the structure by putting the numerator as the value inside of the structure and the denominator is the sum of the layer or row inside of the structure.

VI. Referencs

[1]E. W. Weisstein, "Pascal's triangle," MathWorld, 2002. [Online].Available: https://mathworld.wolfram.com/PascalsTriangle.html.

[2]J. F. Putz, "The Pascal polytope: An extension of Pascal's triangle to N dimensions," The College Mathematics Journal, vol. 17, no. 2, pp. 144-155, 1986.

[3]J. H. Staib and L. H. Staib, "The Pascal Pyramid," Mathematics Teacher: Learning and Teaching PK–12, vol. 71, no. 6, pp. 505–510, 1978.

[4]R. H. Landau, M. J. Páez and C. C. Bordeianu, "Computational Physics: Problem Solving with Python," John Wiley & Sons, pp. 69-84, 2015.

[5]E. W. Weisstein, "RandomWalk1-Dimensional," MathWorld, 2002. [Online]. Available: https://mathworld.wolfram.com/RandomWalk1-Dimensional.html.

[6]M. Cencini et al., "Statistical Mechanics," in A Random Walk in Physics, Springer, Cham, 2021.

[7]Bodyshots, "Pascal [Source Code]," GitHub, 2022. [Online]. Available: https://github.com/Bodyshots/pascal/blob/main/Pascal3D/pascal_3d.py.