Intro Math Problem Solving
October 24

Arrays: Review + New

The Normal Distribution

A Random Walk in 1D

A Random Walk in 2D

References

Chapter 6, Section 2 of our textbook discusses these topics, and can be useful for comparison and background to these notes.

"Insight Through Computing" is available as an ebook on the library web site, and "chapter6.pdf" is also in today's Canvas folder.

Brian Hayes has an interesting article on this topic, called "How to Avoid Yourself", available in today's Canvas folder as "self_avoidance.pdf".

Vectors: Arrays of Numbers

Row and Column Vectors

Because this is an important distinction in linear algebra, every MATLAB list is either a row vector or a column vector.

We aren't doing linear algebra yet, so there are only a few important things to say about the difference.

Initialize a row vector

```
a = [ 1, 2, 3 ];
b = [ 3 4 5 ];  <- Comma separators are optional

c = linspace ( 0.0, 10.0, 101 );
```

In functions with (m,n) input, specify m=1 row and n=anything columns:

```
d = zeros ( 1, 10 );
e = ones ( 1, 5 );
f = rand ( 1, 20 );
g = randn ( 1, 100 );
h = randi ( [ 10, 20 ], 1, 5 ),
```

Initialize a column vector

a = [1; 2; 3]; <- semicolons must be used.

In functions with (m,n) input, specify m=anything row1 and n=1 column:

```
d = zeros ( 10, 1 );
e = ones ( 5, 1 );
f = rand ( 20, 1 );
g = randn ( 100, 1 );
h = randi ( [10,20], 5, 1 );
```

An easy mistake

Many MATLAB functions with (m,n) input will allow you to specify a single value (m) instead.

This looks like you're defining a vector, but you're actually setting up an mxm matrix!

```
d = zeros ( 5 );          <- 5x5 matrix!
e = ones ( 6 );           <- 6x6 matrix!
f = rand ( 7 );           <- 7x7 matrix!
g = randn ( 8 );          <- 8x8 matrix!
h = randi ( [10,20], 9 ); <- 9x9 matrix!
```

Transpose

We don't need to do this yet, but you can convert a row vector to column form, and vice versa, by using the transpose operator, which is simply an apostrophe.

```
x = [ 1, 2, 3];

y = x';
  y is
   [ 1;
    2;
    3 ]

z = y';
  z is [ 1, 2, 3]
```

Recall Vector Functions

If x is a row or column vector, we can:

```
l       = length ( x );
xmax    = max ( x );
xmean = mean ( x ); <- average value
xmin    = min ( x );
xnorm  = norm ( x ); <- sqrt of sum of squares.
xstd    = std ( x );    <- standard deviation
xsum   = sum ( x );
```

Selecting Some of a Vector

Suppose x = [11, 12, 13, 14, 15, 16, 17, 18];

If we type "x(1)", we see "11".

But if we type "x(2:4)" we see "12, 13, 14".

We can use the same "colon notation" that we have used in "for" loops, but now we use it to specify a portion of a vector.

You could even do "x(1:2:8)" to see 11, 13, 15, 17.

Selecting Some of a Vector

If x = [11, 12, 13, 14, 15, 16, 17, 18];

we can use colon indexing to examine or change portions of the vector:

```
x(2:4) = 7;
x = [ 11, 7, 7, 7, 15, 16, 17, 18 ];

x(4:5) = x(4:5) + 10;
x = [ 11, 7, 7, 17, 25, 16, 17, 18 ];

x(6:8) = x(1:3);
x = [ 11, 7, 7, 17, 25, 11, 7, 7 ];
```

Using Logic on Vectors

How many entries of X are greater than 1? We can answer this with a for loop.

But another way creates a logical vector.

```
x = [ 1.2, 0.7, 2.3, 1.5, -1.0 ];

is_greater_than_1 = ( x > 1 );
```

is_greater_than_1 will be [1, 0, 1, 1, 0];

```
how_many = sum ( is_greater_than_1 );
```

how_many will be 3.

OR / AND for Vectors

We have been using || and && for OR and AND operations.

However, just like the DOT operators, we have to do something slightly different when working with vectors.

We have to use | instead of ||, and & instead of &&, when making logical expressions involving vectors!

```
i = ( 0.0 < x & x < 1 )

j = ( x == 0 | y == 0 )
```

More Logic Examples

```
x = [ 1.2, 0.7, 2.3, 1.5, -1.0 ];

small = ( abs ( x ) <= 1 );  small is [0,1,0,0,1];
positive = ( 0.0 < x );      positive is [1,1,1,0];
between12 = ( 1 <= x & x <= 2 );
                             between12 is [1,0,0,1,0].
```

.

A practical example

In a tournament, a loss is -1, a win is +1. Our score is the sum of wins and losses. How many times were we exactly even (score=0)?

score = [0,1,2,1,0,-1,0,-1,0,1];

i = (ahead == 0)

i is [1,0,0,0,1,0,1,0,1,0];

times = sum (i) = 4 times in the tournament.

FINDing Values

Instead of asking whether each entry of a vector satisfies some condition, we might want a list of the locations of all such entries. The FIND command will do this.

```
x = [ 1.2, 0.7, 2.3, 1.5, -1.0 ];
i = find ( x > 1 );
```

Then i is [1, 3, 4] because x(1), x(3) and x(4) are greater than 1.

Moreover, typing "x(i)" will print exactly those values:
1.2, 2.3, 1.5

When you index a list, the index itself can be a list!

More FIND Examples

```
x = [   1.0347,   0.7269, -0.3034,  0.2939, -0.7873,
        0.8884, -1.1471, -1.0689,  -0.8095, -2.9443 ]
```

```
i = find ( abs ( x ) <= 1 );        i = [ 2, 3, 4, 5, 6, 9 ];
j = find ( 0.0 < x );               j = [ 1, 2, 4, 6 ]
k = find ( 1 <= x & x <= 2 );       k = [ 1, 7, 8 ]
l = find ( x == 0 );                l = [];
```

.

Using FIND

You can use FIND to find the parts of a vector you are interested in, and then print, or sum, or otherwise work with just that set.

```
x = [   1.0347,   0.7269, -0.3034,  0.2939, -0.7873,
        0.8884, -1.1471, -1.0689,  -0.8095, -2.9443 ]

j = find ( 0.0 < x );              j = [ 1, 2, 4, 6 ]

>> x(j)

ans =

   1.0347
   0.7269
   0.2939
   0.8884
```

Quiz

x = [0, 1, 2, 1, 2, 3, 2, 1, 0, -1, -2, -1, 0, 1];

Without using FOR or IF/ELSE, write MATLAB expressions that:

A) count the number of x values equal to 1;

B) list the locations of x values equal to 1;

C) count the number of x values equal to 0;

D) list locations of x values less than 0, print them;

E) count x values equal to 1 OR 3;

F) count x values NOT equal to 1;

Quiz Answers

A) count the number of x values equal to 1:

```
i = ( x == 1 );     num = sum ( i );
```

B) list the locations of x values equal to 1:

```
j = find ( x == 1 );
```

C) count the number of x values equal to 0:

```
i = ( x == 0 );     num = sum ( i );
```

D) list the locations of x values less than 0 and print:

```
j = find ( x < 0 ); x(j)
```

E) count x values equal to 1 OR 3:

```
i = ( x == 1 ) || ( x == 3 );     num = sum ( i );
```

F) count x values NOT equal to 1:

```
i = ( x ~= 1 );     num = sum ( i );
```

Adding Values to a Row Vector

If X is a row vector, we add entries to it separated by COMMAS:

```
x = [ 1, 2, 3];
y = [ 4, 5 ];

x = [ x, 99 ];      x is now [1,2,3,99];
x = [ 48, x ];      x is now [48,1,2,3,99];
x = [ x, y ];       x is now [48,1,2,3,99,4,5]
y = [ y, y, y ];    y is now [4,5,4,5,4,5]
```

Adding Values to a Column Vector

If X is a column vector, we add entries to it, separated by SEMICOLONS;

```
x = [ 1; 2; 3];
y = [ 4; 5 ];

x = [ x; 99 ];      x is now [1;2;3;99];
x = [ 48; x ];      x is now [48;1;2;3;99];
x = [ x; y ];       x is now [48;1;2;3;99;4;5]
y = [ y; y; y ];    y is now [4;5;4;5;4;5]
```

Adding Values by Index

Another way to add values to a list is to pick an index (location) and store the new value there.

Suppose we have a fever, and from time to time we take our temperature. We could plan to store these values in a list called "temp", but we don't know how many times we will take the measurements.

Then we need two things, a list "temp" and a current index "k", which starts at 0. Every time we take a measurement, we do two things:

```
k = k + 1;
temp(k) = current thermometer reading.
```

MATLAB allows us to create a list that grows as we need it.

Growing List Example

```
temp(1) = 98.6;
length ( temp )
temp

temp(2) = 99.1;
length ( temp )
temp

temp(4) = 101.1
length ( temp )
temp                <- What will be in temp(3)?
```

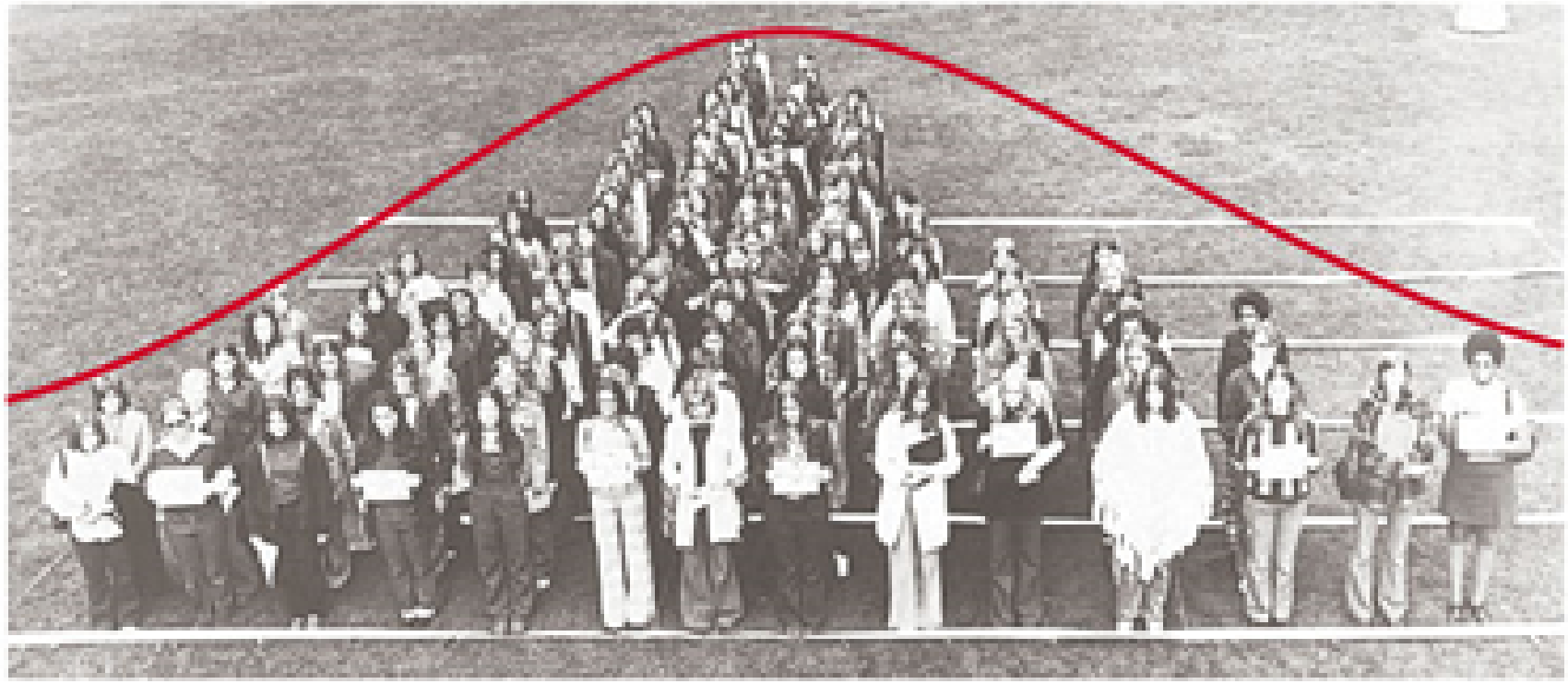
Growing List

We will need this growing list example to follow a random walker who moves from one numbered location to another.

Each time the walker takes a step, we want to add the new location to our growing list.

When we are done, we will have a list of all the places the walker visited.

The Normal Distribution



The Normal Distribution

We have already seen that MATLAB has a random number generator function randn() which generates normal, rather than uniform, random numbers.

To begin with, let's compute 5 samples of each type:

x1 = rand(1,5):

0.8147 0.9058 0.1270 0.9134 0.6324

x2 = randn(1,5)

-1.3077 -0.4336 0.3426 3.5784 2.7694

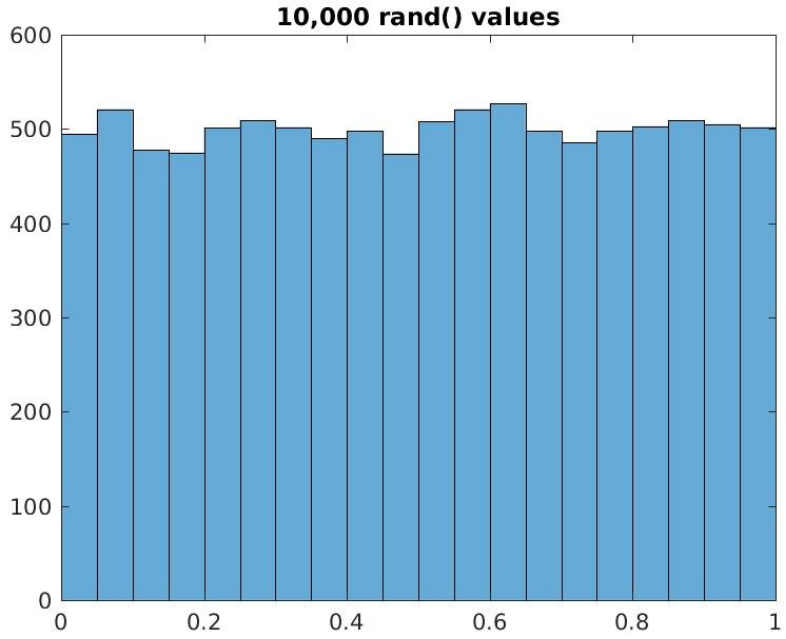
RAND vs RANDN

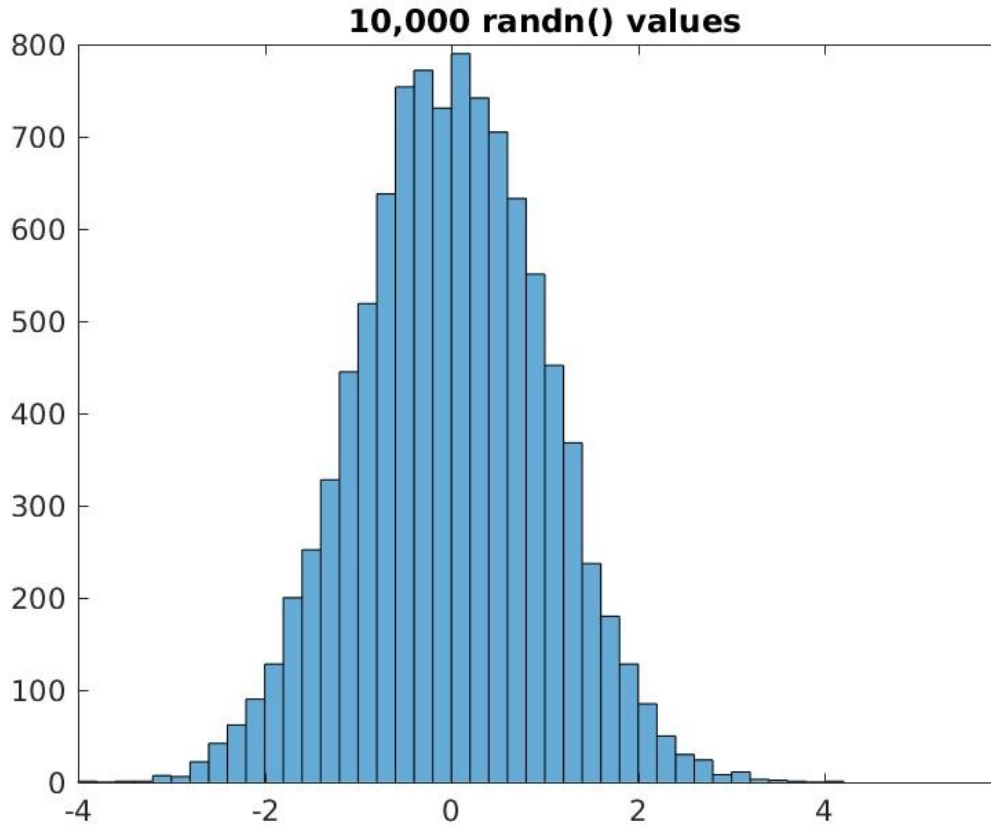
We note that the rand() values are between 0 and 1, while the randn() values are more spread out, and include negative values.

But we see why the word "uniform" is used for rand if we compute a histogram of 10,000 values from each function:

```
x1 = rand(1,10000);  histogram ( x1 );
x2 = randn(1,10000); histogram ( x2 );
```

histogram(x1) vs histogram(x2)





Sampling Different Kinds of Events

Uniform random numbers are useful when describing events for which each outcome is thought to be equally likely. We use randi() when there are only a limited number of choices (integer between a and b), and rand() when there is a range (real number between 0 and 1).

Normal random numbers are useful when there is a most likely or average outcome, and the likelihood of other outcomes depends on being close to the average.

The Normal Distribution

In the mathematical theory of probability, there is a function, called the normal probability distribution function or normal pdf, which indicates the likelihood of any event.

We think of an "event" as a number x.

"mu", the mean, is the most likely outcome.

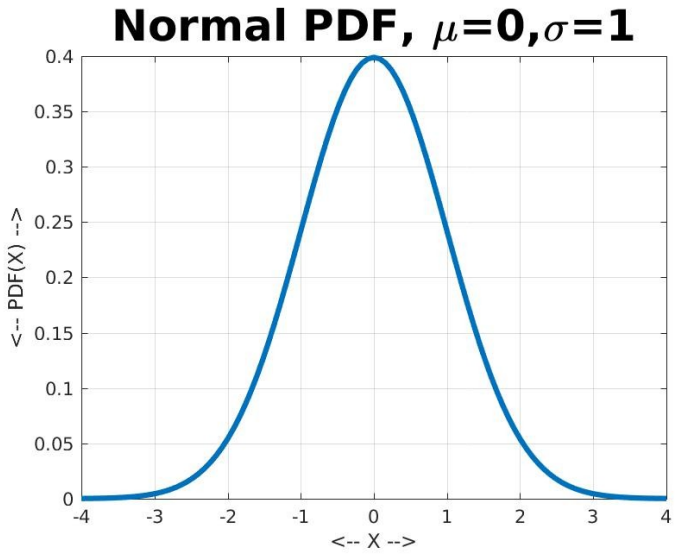
"sigma", the standard deviation, measures how far away from the mean most outcomes will be.

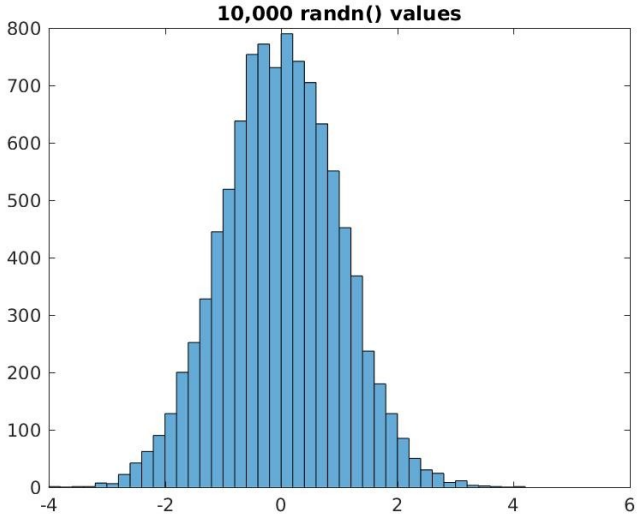
The most common values are mu=0, sigma=1.

The Normal Distribution

The Normal PDF is then:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$





normal_pdf.m

```
x = linspace ( -4, +4, 101 );
y = normal_pdf ( x, 0, 1 );
plot ( x, y, 'linewidth', 3 )
grid on
xlabel ( '<-- X -->' )
ylabel ( '<-- PDF(X) -->' )
title ( 'Normal PDF, \mu=0,\sigma=1', 'Fontsize', 24 )
print ( '-djpeg', 'normal_pdf.jpg' )

function value = normal_pdf ( x, mu, sigma )
  value = exp ( - ( x - mu ) .^2 / 2.0 / sigma^2 ) ...
    / sqrt ( 2.0 * pi * sigma^2 );
  return
end
```

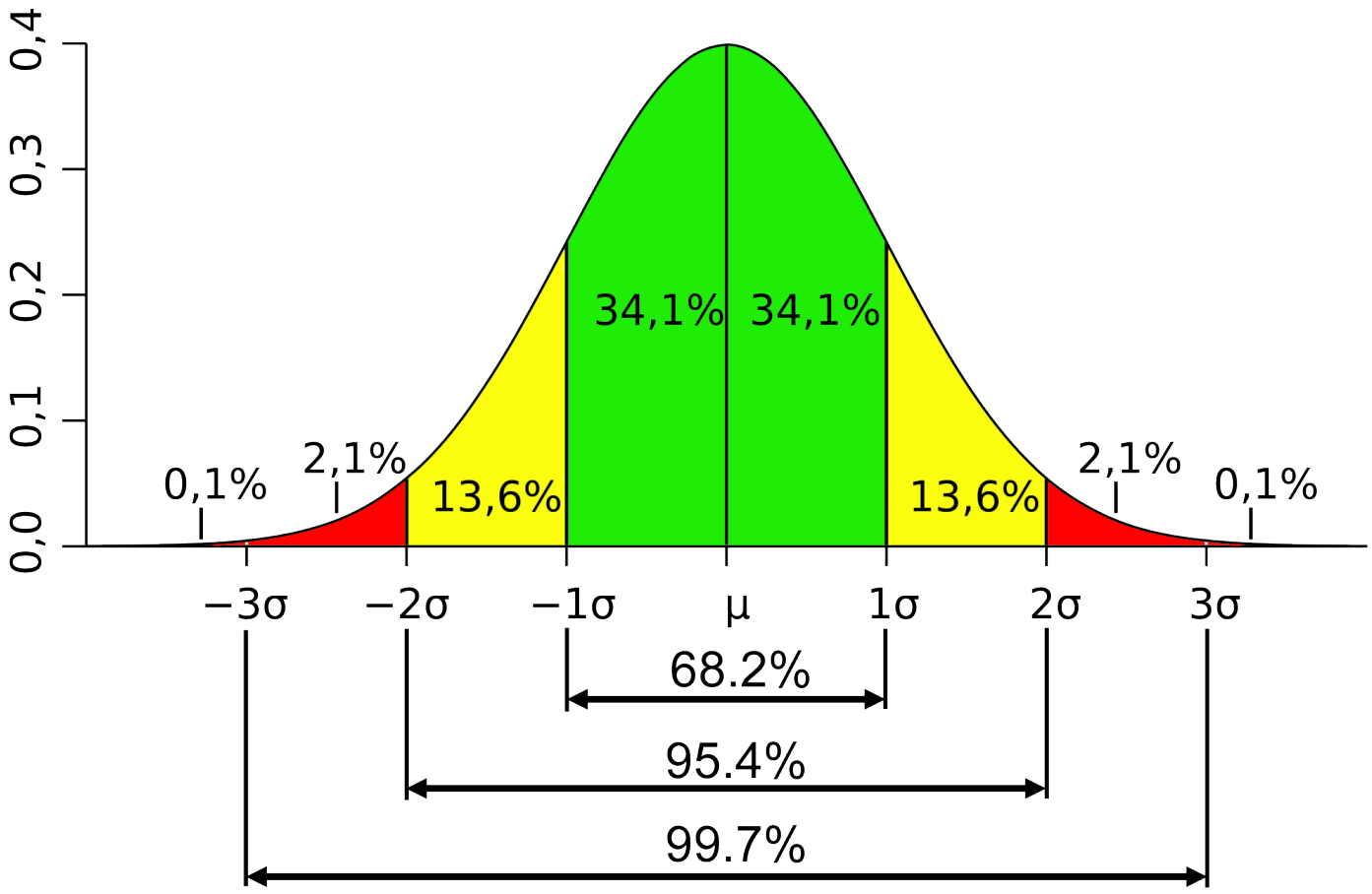
The Standard Deviation

The quantity "sigma", or standard deviation, gives a measure of how spread out the sample values will be from a normal distribution.

In particular, it estimates that about 34% of the values will be within one sigma below the mean, and 34% will be within one sigma above the mean.

We can test this with an experiment.

The Standard Deviation



For RANDN, Mu = 0, Sigma = 1

```
x = randn ( 1, 1000 );

i1 = ( -1.0 <= x & x <= 0.0 );
i2 = (  0.0 <= x & x <= 1.0 );

n1 = sum ( i1 );    ( n1 is 337 )
n2 = sum ( i2 );    ( n2 is 354 )
```

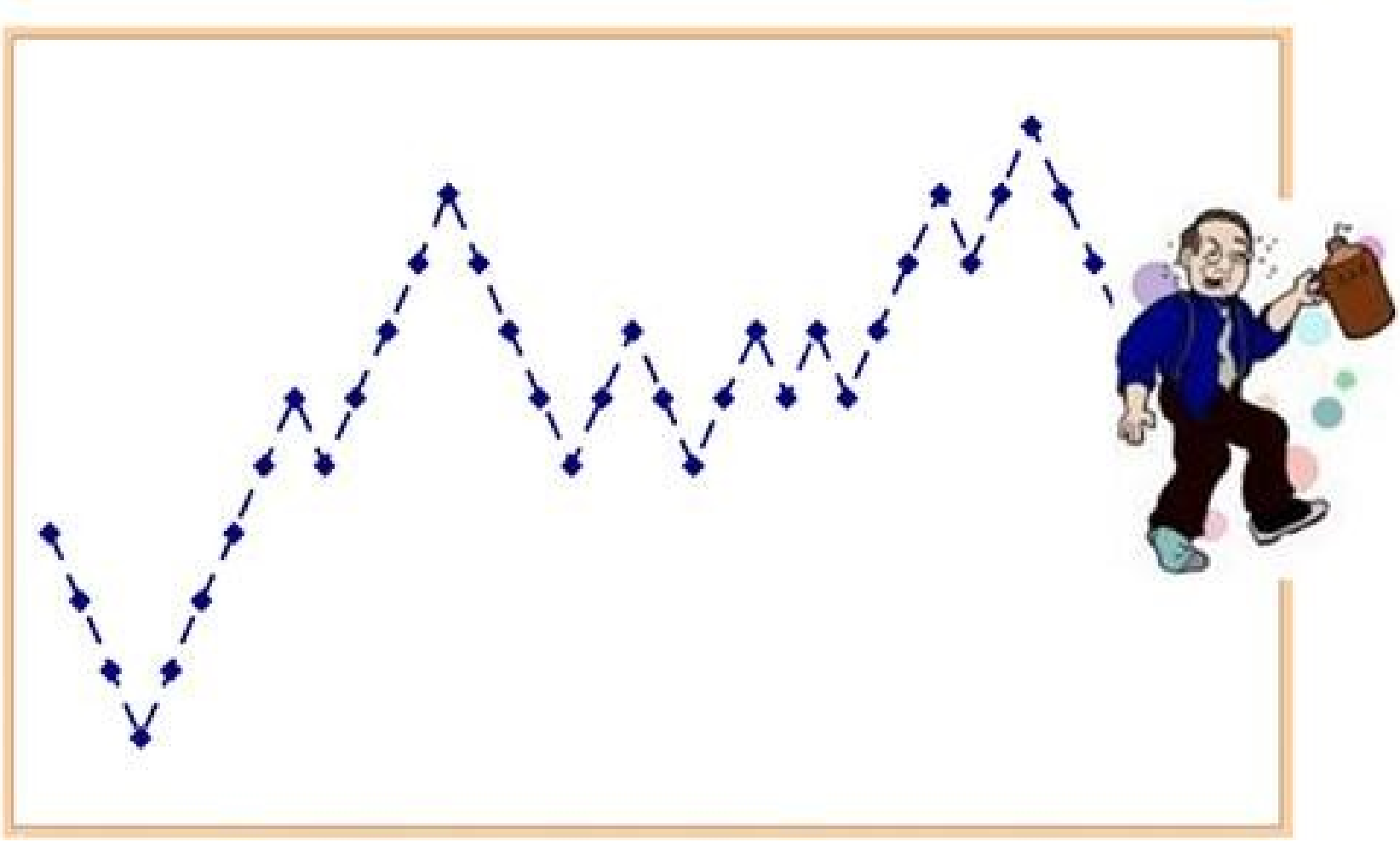
How would we count the items that are between 1 and 2 standard deviations away, and about how many should we expect to find?

Random doesn't mean patternless!

We have discussed the normal pdf because we are about to look at a random process for which some behaviors will start to match the pattern we have seen for normal random numbers.

This means that even when a process is random, there may be features of it which are predictable, or have a pattern, at least if we think in terms of averages.

A Random Walk in 1 Dimension



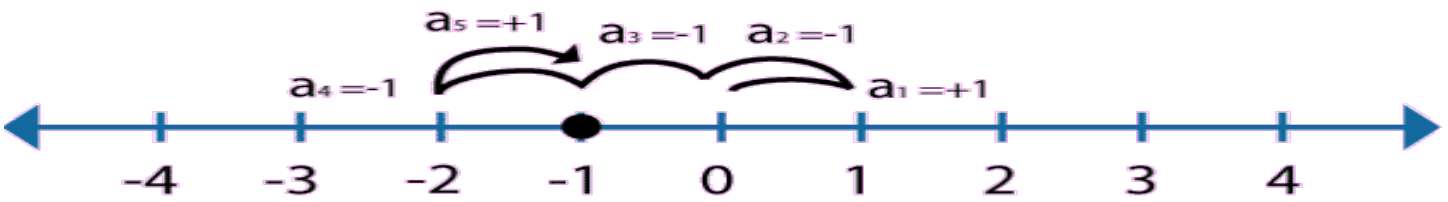
Why Random Walks?

A random walk seems like a peculiar thing to study. However, it models a real physical situation, "Brownian motion", noticed by Robert Brown in 1827, watching grains of pollen suspended in water, that randomly jiggled.

The puzzle wasn't fully solved until Albert Einstein explained this by the collision of the pollen with individual water molecules whose velocities had a random variation.

From this, Einstein was able to show that a particle in Brownian motion would tend to drift away from its original position with a predictable variation.

Rules for a Random Walk in 1D



Consider a sidewalk of numbered squares; perhaps numbered x=-N to x=+N;

We put a walker on some square, perhaps the one labeled "x=0";

The walker takes a series of steps, each randomly chosen to the left (x=x-1) or right (x=x+1).

The walk stops at the boundary: (x=-N or x=+N).

We keep track of each step in an array "track".

We need a "growing list" for random walk

In a "random walk", we start at a given location, and then make random moves until we reach a boundary or a goal.

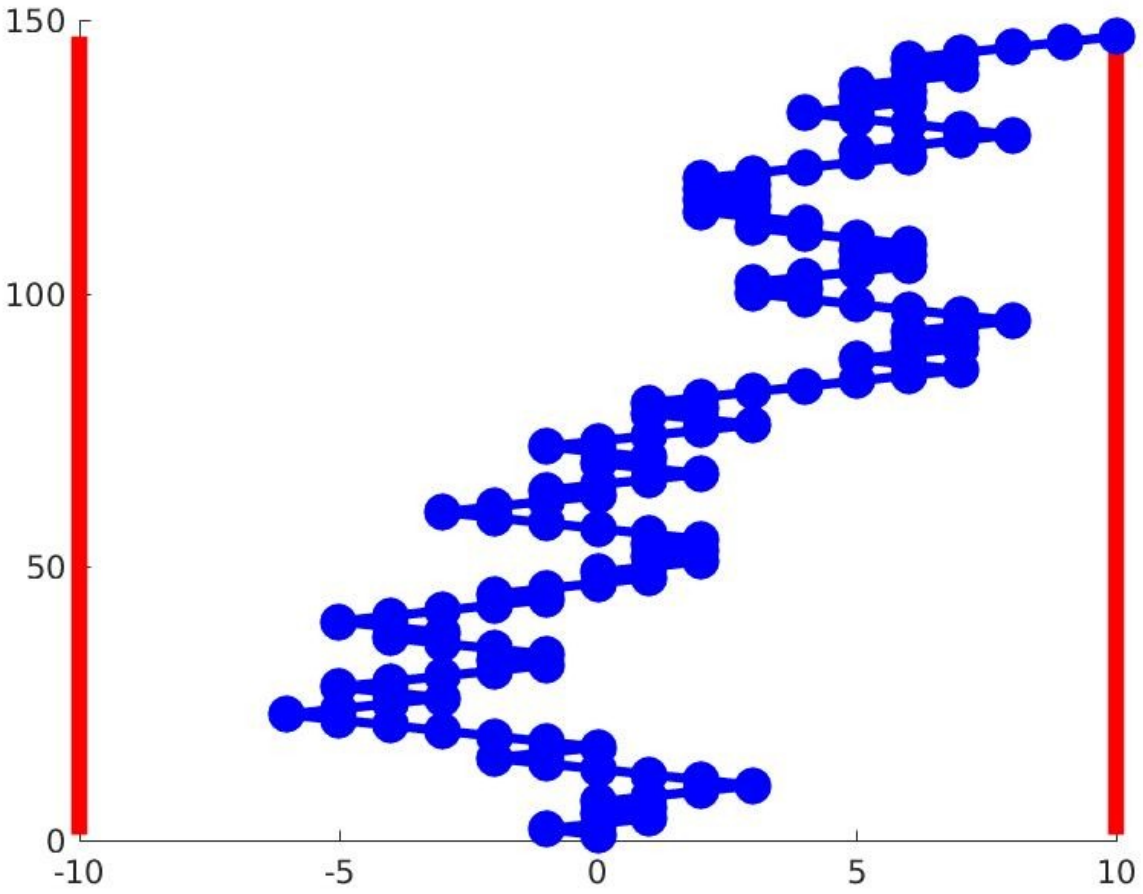
If we wish to list the steps involved in the random walk, we don't know in advance how many steps are involved.

MATLAB allows us to start a list, and just keep adding one more element to it, until we decide we have completed the walk.

Simulate 1 Random Walk

```
function track = walker_1d ( n )       <- User specifies N, the size of "sidewalk"

  x = 0;
  k = 1;  track(k) = x;       % <- Begin the list

  while ( abs ( x ) < n )     % <- Stop at boundary.

    i = randi ( [ 1, 2 ] );     % <- Randomly choose 1 or 2.

    if ( i == 1 )
      x = x - 1;
    else
      x = x + 1;
    end

    k = k + 1;  track(k) = x;  % <- Add this step to list.

  end

  return
end
```

Random Walk of 150 Steps



How Was That Plot Drawn?

How did I make the plot of the 1D random walk?

I know there were 150 steps.

I can draw red barrier lines.

If TRACK(I) = X, I want to draw a blue dot at location (X, I).

I want to draw a line connecting each consecutive pair of dots.

Could you do this?

Was 150 Steps Typical?

In the previous example, it took 150 steps for the random walker to get 10 units away from the start.

But it's possible to reach that spot by taking just 10 steps. If we try this experiment many times, someone should actually make it that fast.

A single random experiment gives us an answer, but it won't be repeatable. The answer we can look for, however, is what the typical or average number of steps might be.

To do that, we can simply try a reasonably large number of experiments, record the length of each, and report the average.

Seek Average for N=10 random walk

```
function average = walker_1d_average ( n, m )

%% WALKER_1D_AVERAGE averages the length of a random walk on [-N,+N].
%
%  It is assumed the walker starts at 0, and randomly steps left or
%  right until reaching -N or +N.  The number of steps taken is the
%  length of the walk.
%
%  This program takes M random walks, and reports the average length.
%
  average = 0.0;

  for i = 1 : m
    track = walker_1d ( n );
    average = average + length ( track ) - 1;
  end

  average = average / m;

  return
end
```

Estimated Average for N = 10 is about 97

```
n = 10;
m = 1000;
average = walker_1d_average ( n, m )
fprintf ( 'Averaging over %d walks\n', m );
fprintf ( 'The sidewalk runs from %d to %d\n', -n, n );
fprintf ( 'The average walk takes %g steps\n', average );
```

```
Averaging over 1000 walks
The sidewalk runs from -10 to 10
The average walk takes 96.9940 steps
```

How Does the Average Depend on N?

When N = 10, the average number of steps is about 97, or almost 100, which happens to be 10^2.

We might hope that this relationship is correct, and for any value of N, the average number of steps is N^2.

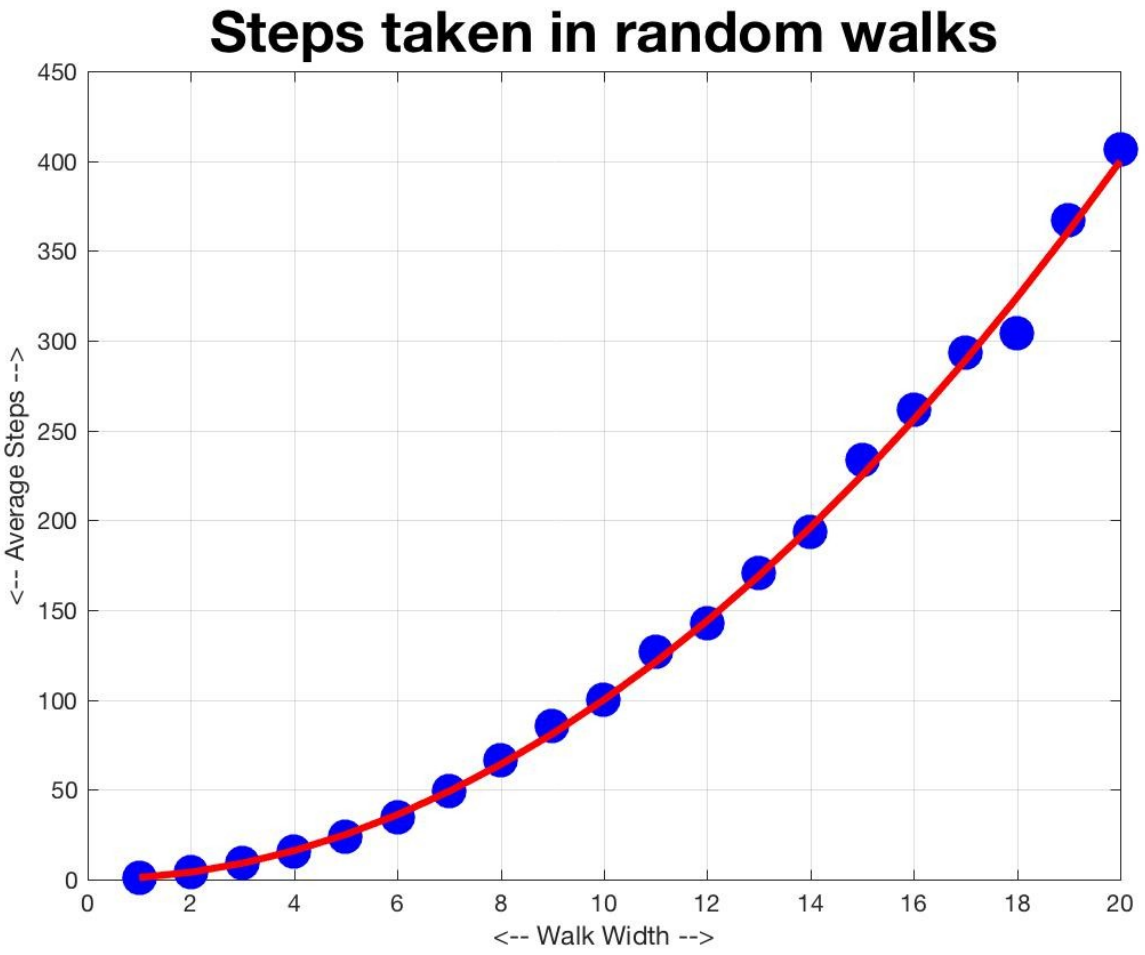
Before trying to prove an idea like this, it's worth trying to see if the evidence supports it. So we will look at a range of N values and estimate the walk lengths.

walker_1d_averages.m

```
m = 1000;
fprintf ( 'Averages are based on %d trials.\n', m );

fprintf ( '\n' );
fprintf ( '   N    Average length\n' );
fprintf ( '\n' );

for n = 1 : 20
  average(n) = walker_1d_average ( n, m );
  fprintf ( '  %2d  %f\n', n, average(n) );
end
x = 1:20;

plot ( x, average, 'b.', x, x.^2, 'r-', 'LineWidth', 3, 'MarkerSize', 50 );
grid on
xlabel ( '<-- Walk Width -->' );
ylabel ( '<-- Average Steps -->' );
title ( 'Steps taken in random walks', 'Fontsize', 24 );
print ( '-djpeg', 'walker_1d_averages.jpg' );
```

Average "Escape" Time is N^2

N	Average length
1	1.000000
2	3.990000
3	8.914000
4	15.970000
5	25.732000
6	36.710000
7	50.714000
8	64.234000
9	78.724000
10	99.902000
11	119.124000
12	150.124000
13	167.358000
14	188.682000
15	229.628000
16	258.818000
17	289.834000
18	331.768000
19	379.750000
20	399.604000



Watch Many Walkers Over Time

Instead of watching one walker, consider 1,000, all starting at 0. As time passes, they spread out. Although their steps are random, the overall pattern is regular, and can be approximated by the normal probability function:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Bin That Data!

We don't want to plot 1000 walker tracks. We won't see any pattern.

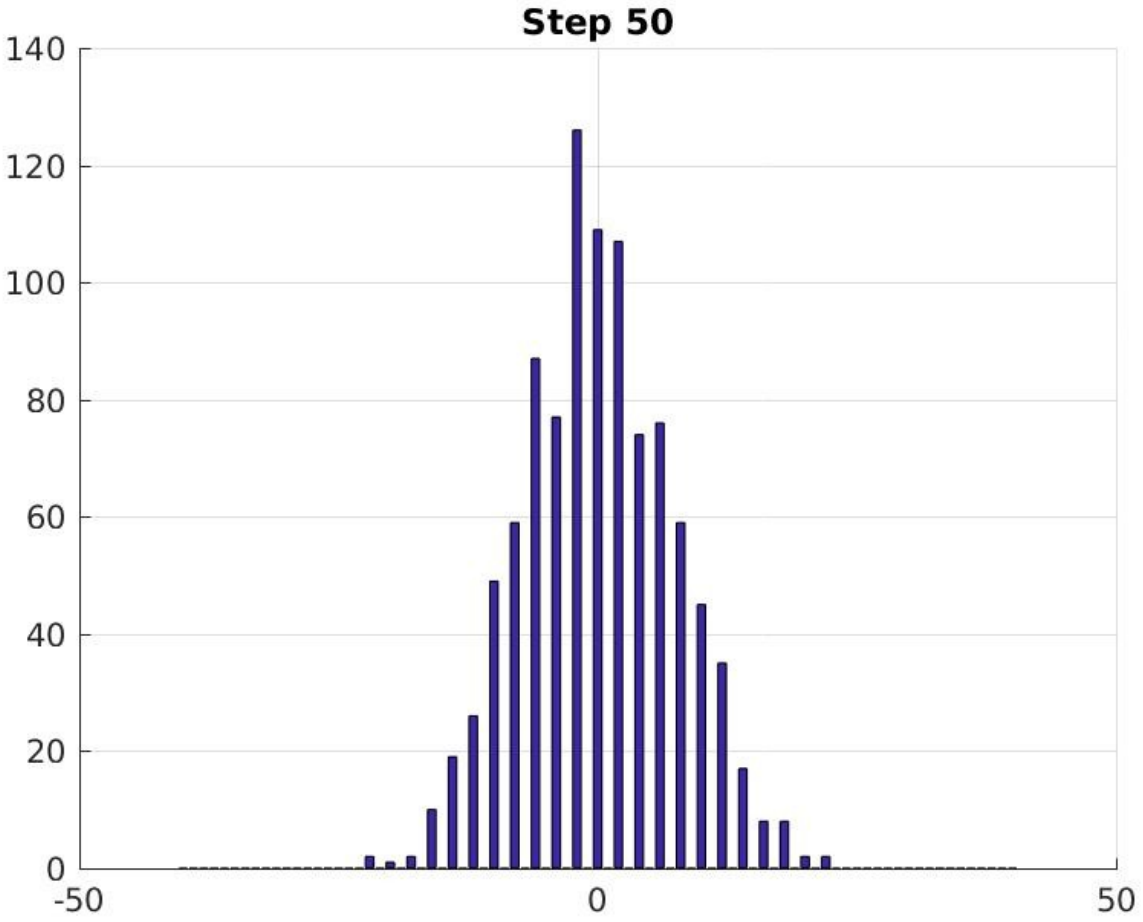
But suppose instead we make boxes (technically called bins) labeled -41 through +41, and, at any time, count how many walkers were occupying that position.

At the first time, all the walkers are at position 0.

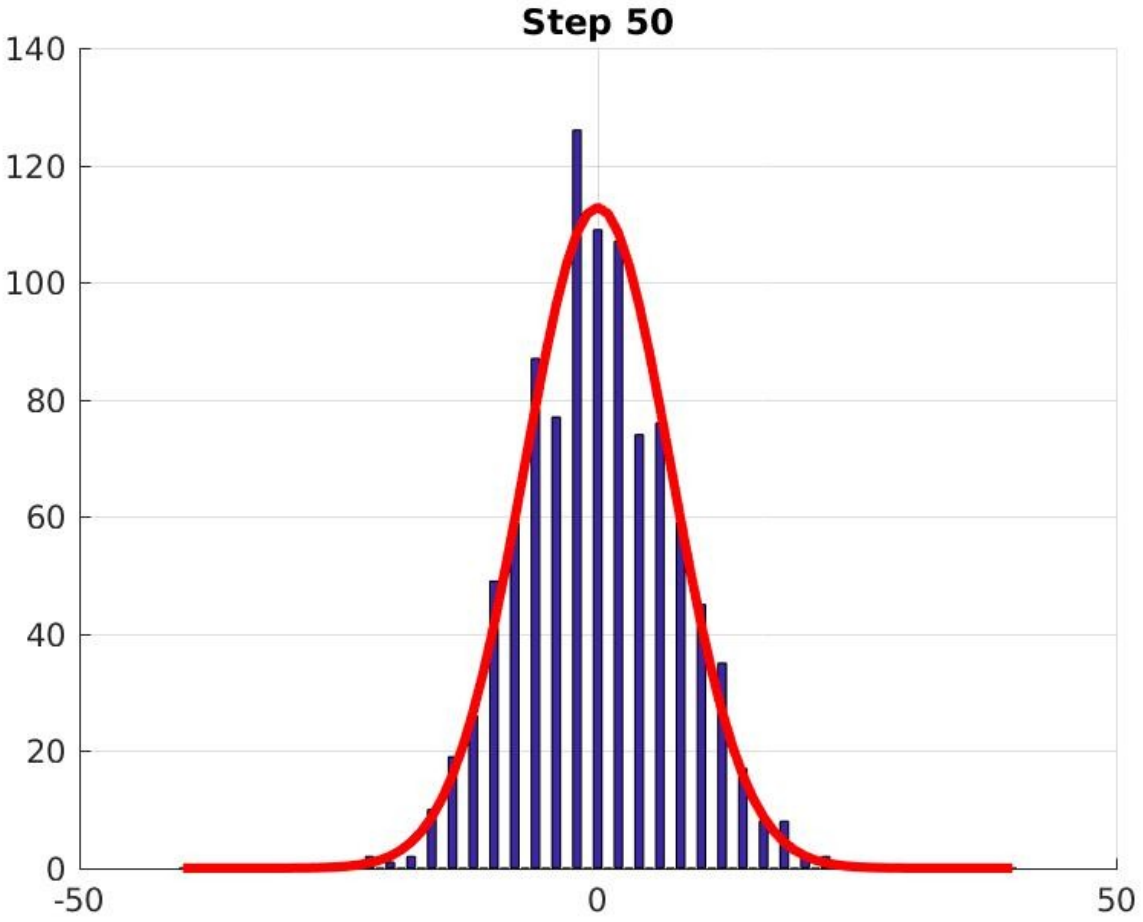
At the second time, about half are at 1, and half at -1.

After that, we assume that they gradually spread out, and we may see a pattern in this behavior.

1000 Walkers after 50 Steps



The Normal Distribution



How was that plot drawn?

Each walker can be in any location from -41 to +41.

Each entry of my "bins" array counts how many walkers are at a specific location.

bins(1) counts the walkers at -41:

```
i = ( x == -41 )
bins(1) = sum ( i )
```

Once I have the bins set, I call MATLAB's bar() plotting command:

```
bar ( -41:41, bins );
```

Walkers track Locations

If we have 1 walker, it makes sense to create a list "track()" that lists the sequence of locations visited. At any one time, we know a single number, "x".

If we have 1000 walkers, then we might create 1000 lists called "track1" ... "track1000". These lists will have different lengths. At any one time, we will have up to 1000 active walkers to keep track of.

Following a large number of walkers can become a difficult task to manage!

Locations Count Walkers

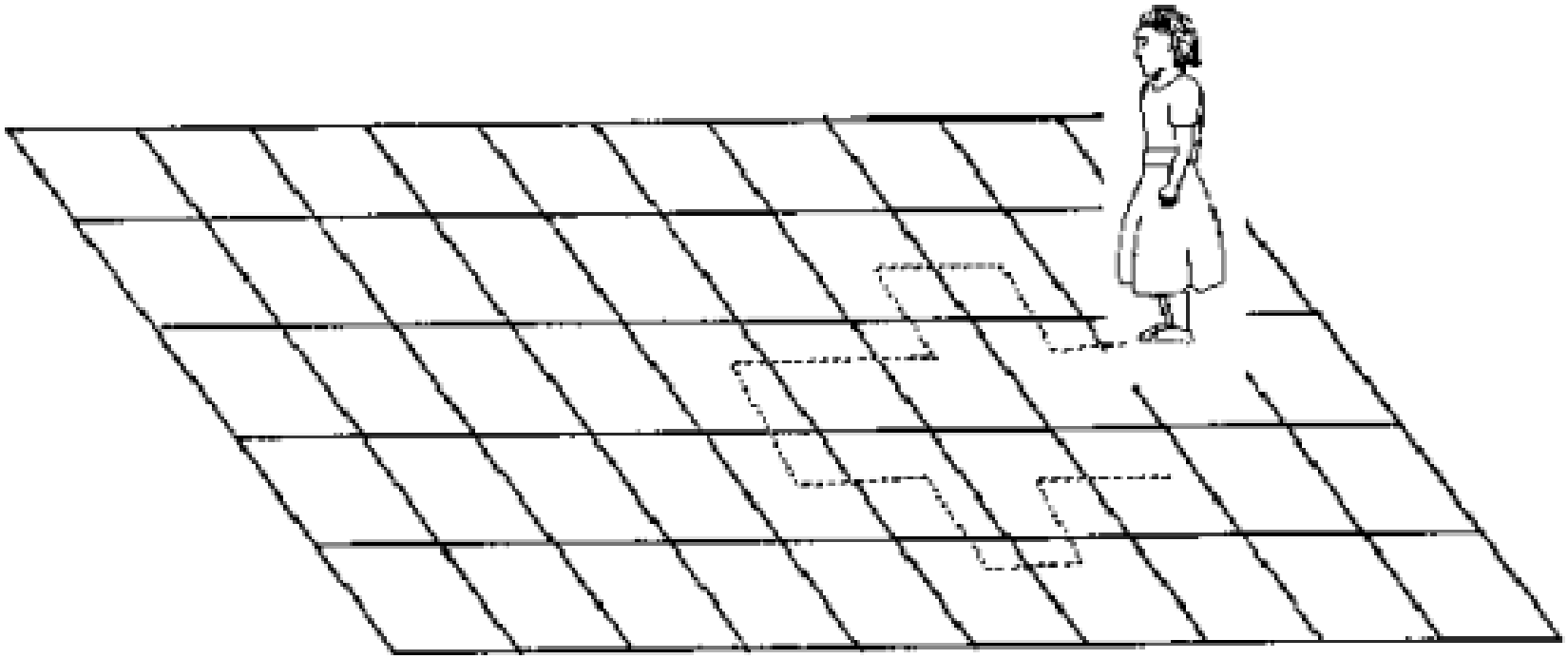
Instead of each walker keeping a sort of diary of where it is, we could have each location report how many walkers are there.

In the plots we looked at a moment ago, there were 1000 walkers, but 81 locations.

If we set up a list of length 81, we can count the number of walkers in location -40, -39, -38, ..., -1, 0, +1, ..., 38, 39, 40.

In other words, the bins we used for plotting could perhaps also be used for computing.

A Random Walk in 2D



Rules for a Random Walk in 2D

Consider a checkerboard of numbered squares, indexed by (X,Y). -N <= X,Y <= +N.

We put a walker on some square, perhaps the one labeled "(0,0)";

The walker repeatedly chooses the next step at random: north, south, east or west;

To step North, for instance, we move from (X,Y) to (X,Y+1).

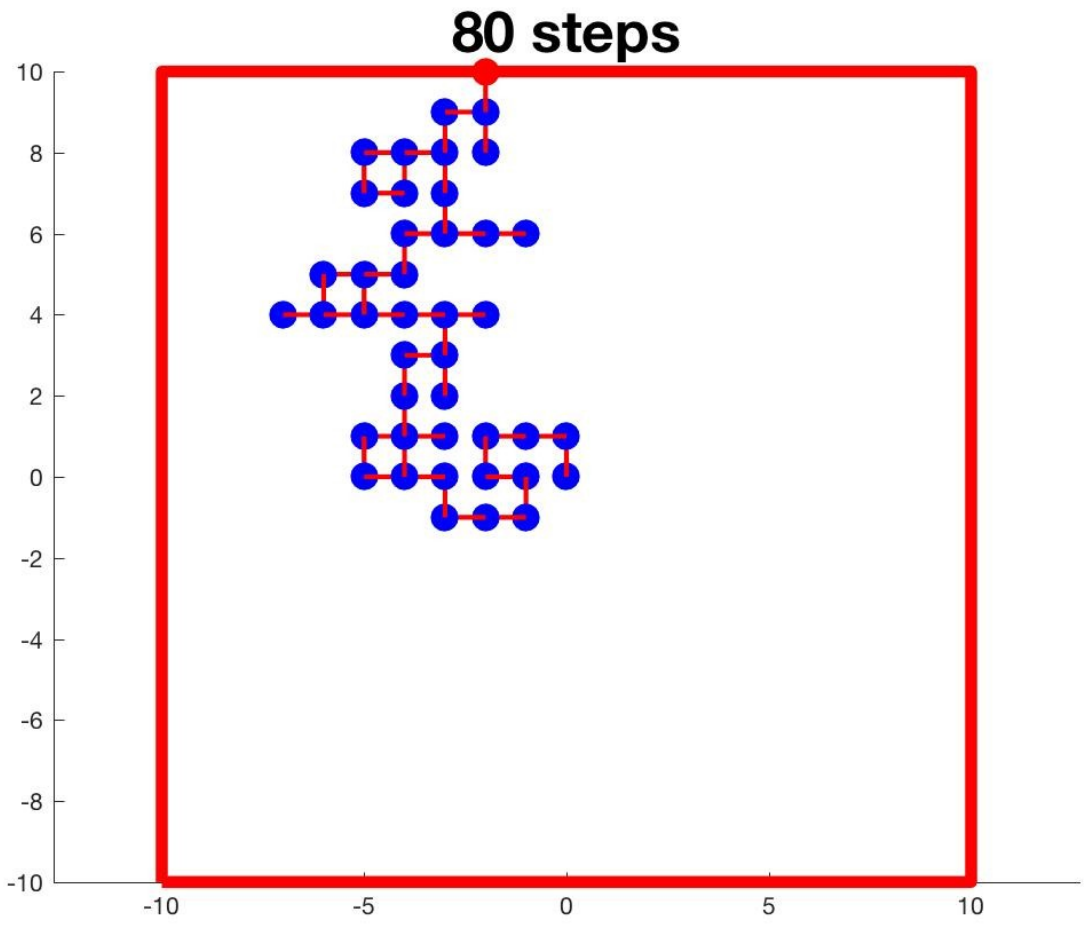
The walk stops at the boundary, where X or Y reaches the value -N or +N.

We keep track of each step in arrays (growing lists) "xtrack" and "ytrack".

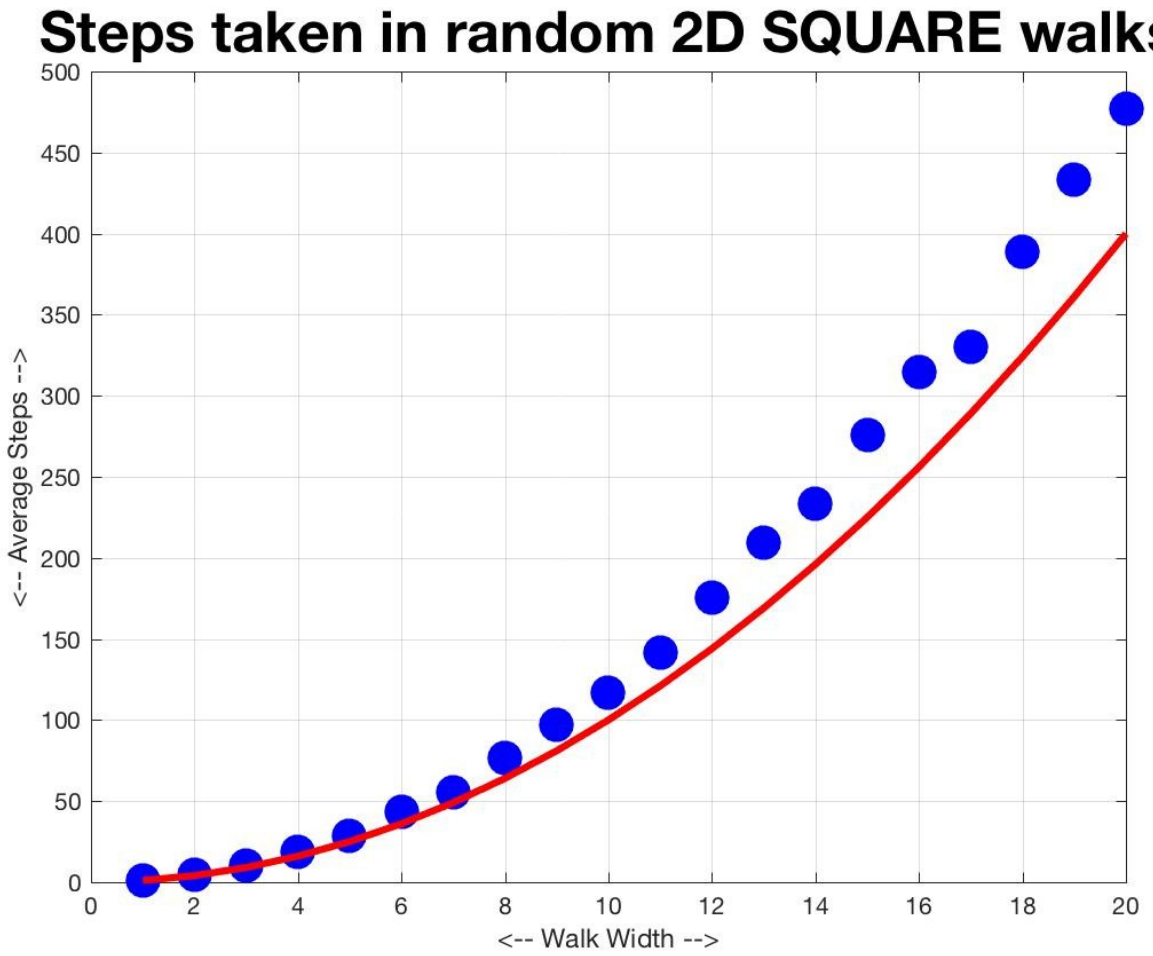
walker_2d.m

```
function [ xtrack, ytrack ] = walker_2d ( n )

  x = 0;   y = 0;
  k = 1;   xtrack(k) = x;  ytrack(k) = y;

  while ( abs ( x ) < n && abs ( y ) < n )

    i = randi ( [ 1, 4 ] );

    if ( i == 1 )
      y = y + 1;
    elseif ( i == 2 )
      y = y - 1;
    elseif ( i == 3 )
      x = x + 1;
    else
      x = x - 1;
    end

    k = k + 1;    xtrack(k) = x;   ytrack(k) = y;

  end

  return
end
```

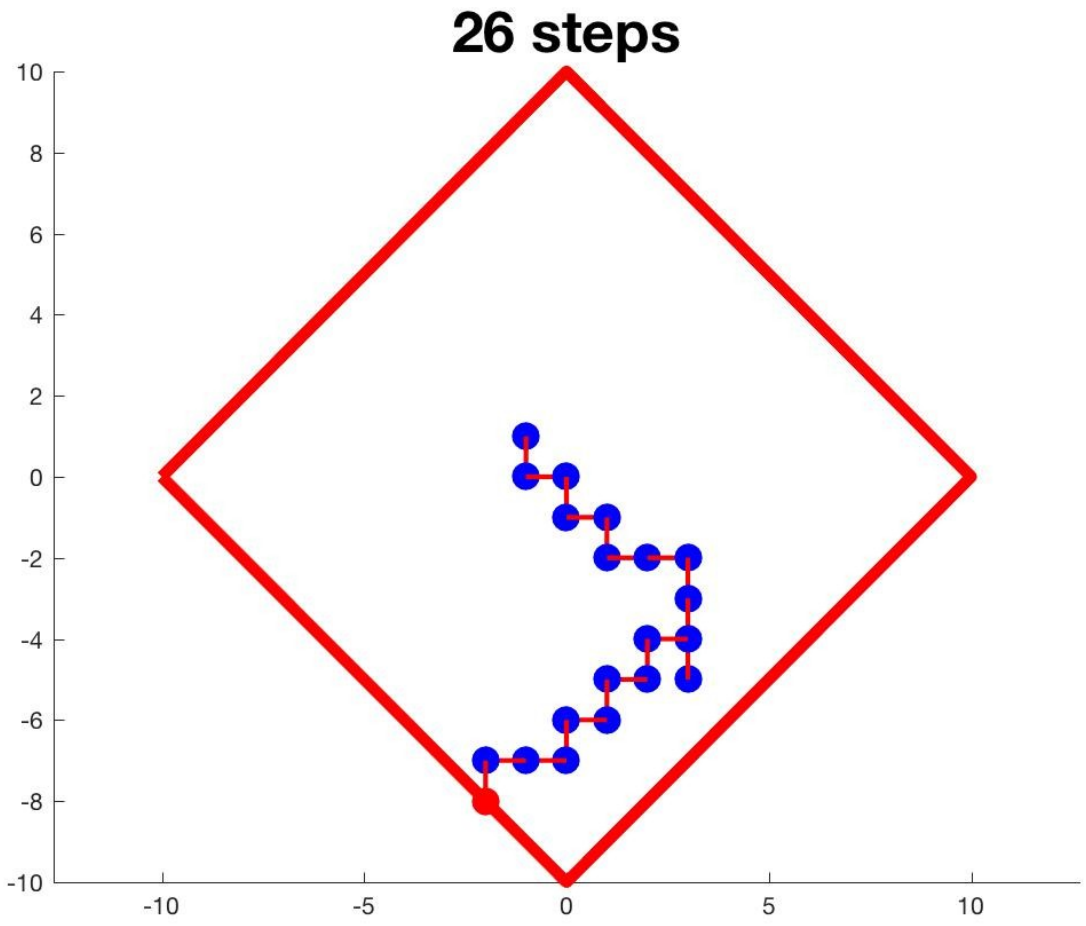
A Typical Random Walk in the Square



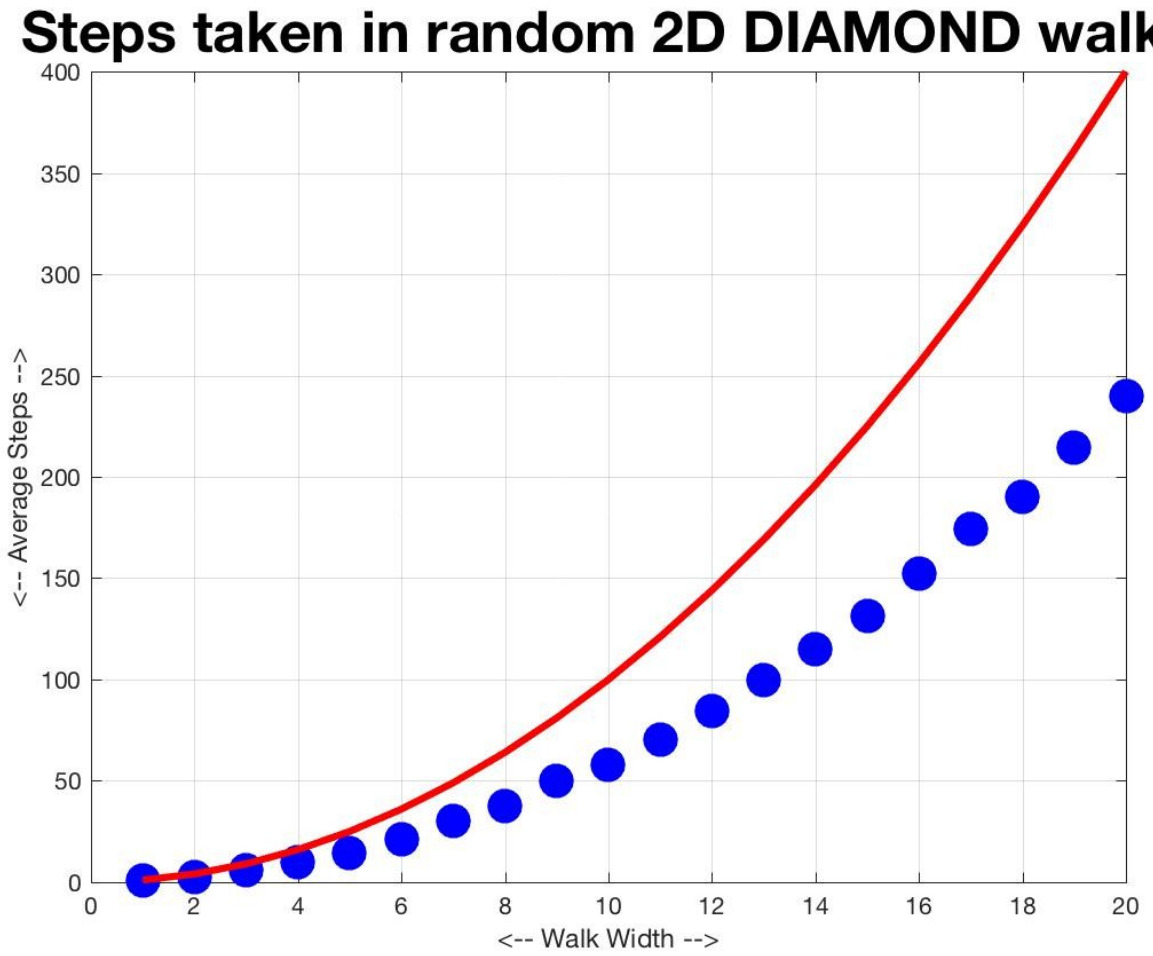
Step Lengths Pull Away From N^2



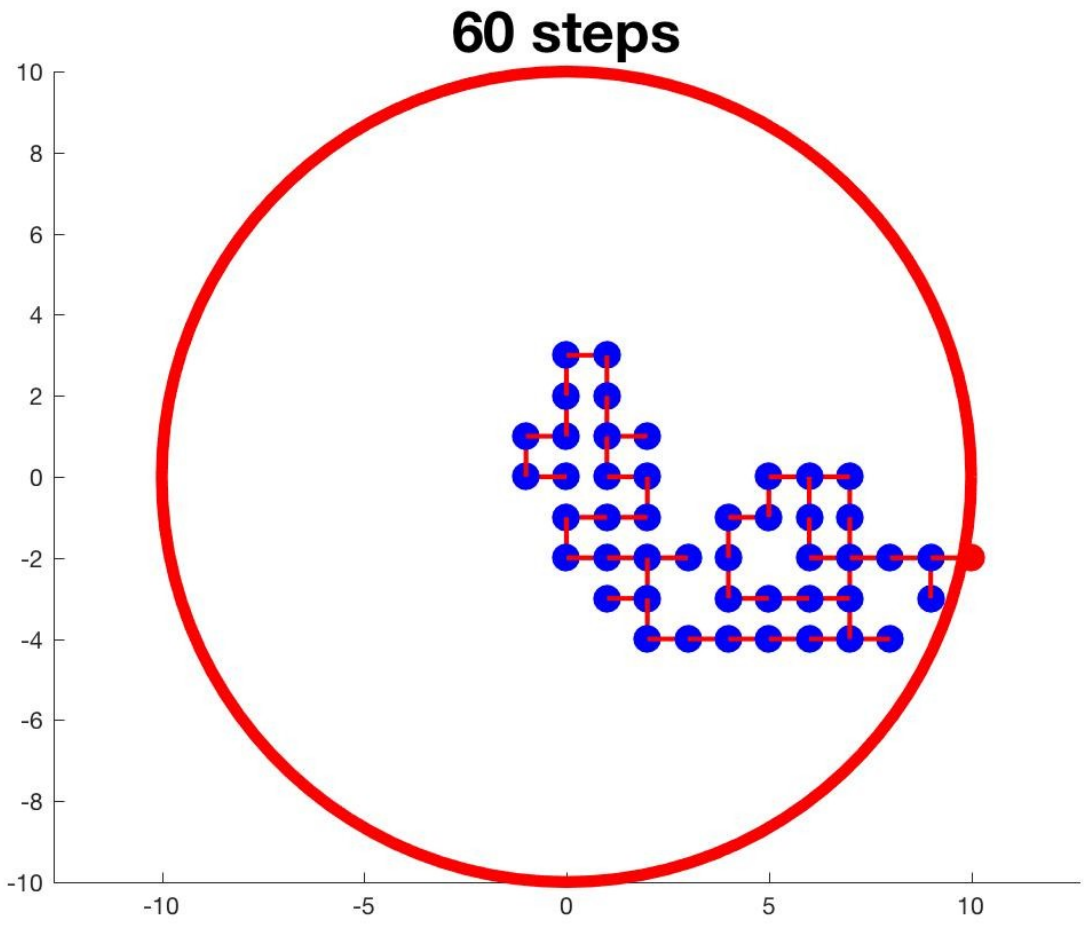
Try Random Walks in Diamond



Diamond Walk Lengths Go Below N^2



Try Walks in Circle



Walk Lengths Closely Match N^2