Intro Math Problem Solving
October 26

Random Walk in 2D

Working With Data in a 2D Matrix

Storing Mario in a 2D Matrix

A Cellular Automaton

The Game of Life

The Mandelbrot Set

Lights Out

Homework #8

Office Hours

Friday (tomorrow): VERY SHORT

Newman Library, 2:00-2:30

Monday: Regular

Torgersen 3050, 2:00-4:00

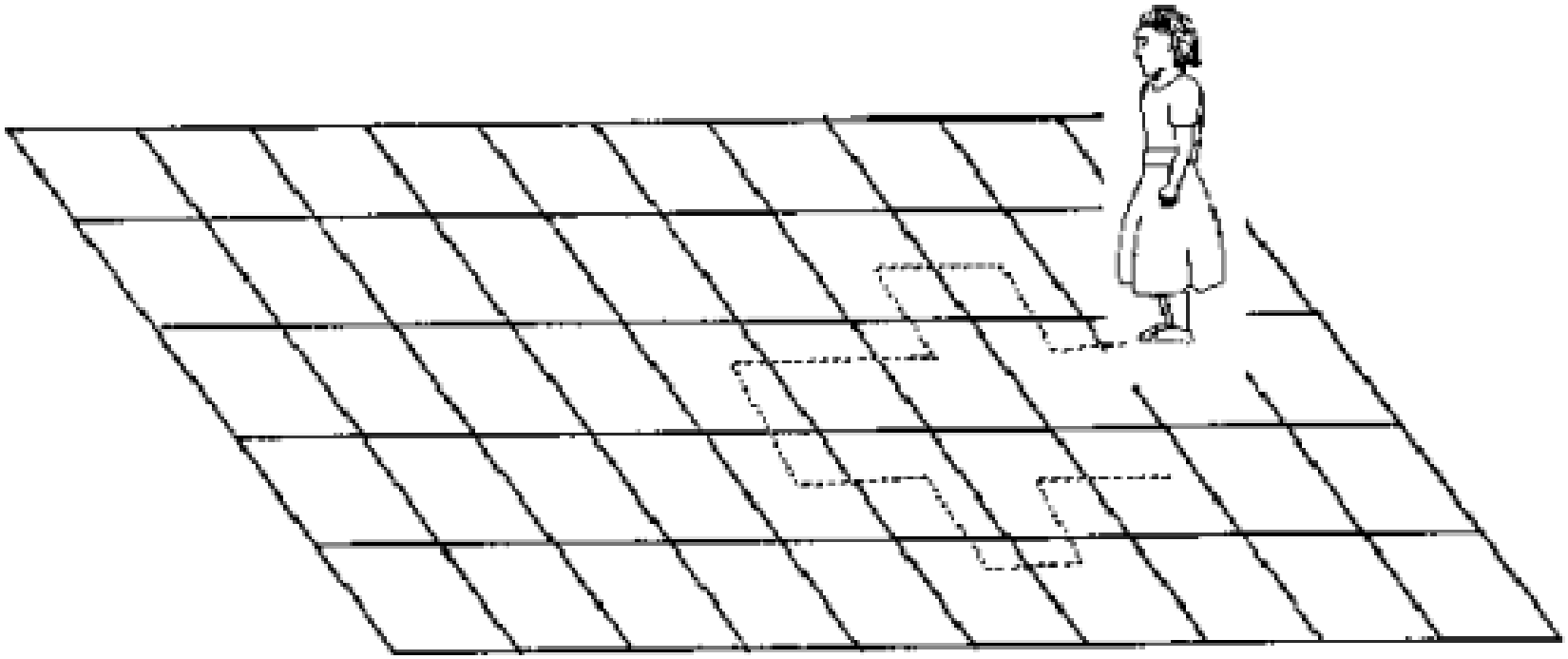
Wednesday: A little short

Torgersen 3050: 2:00-3:30

Friday: Regular

Newman Library, 2:00-3:30

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).

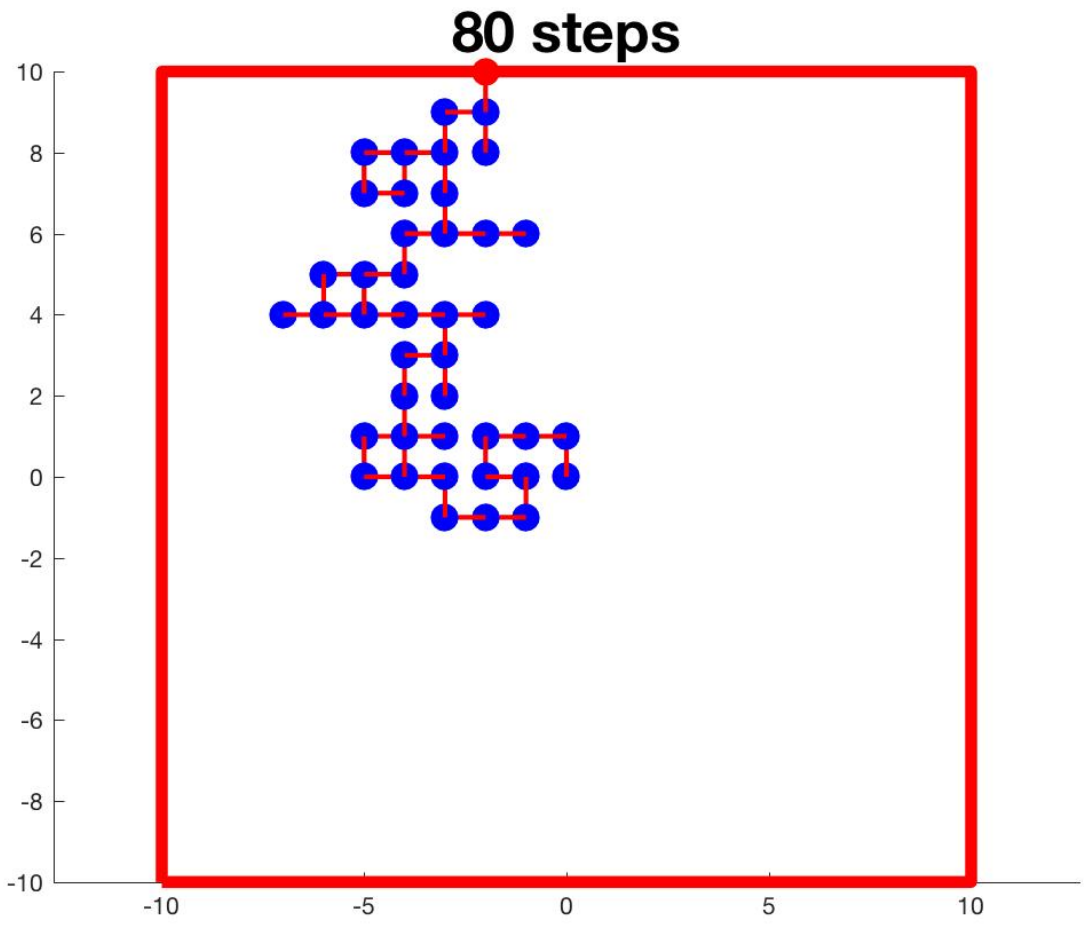
The walk stops at the boundary, where X or Y reaches the value -N or +N.

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

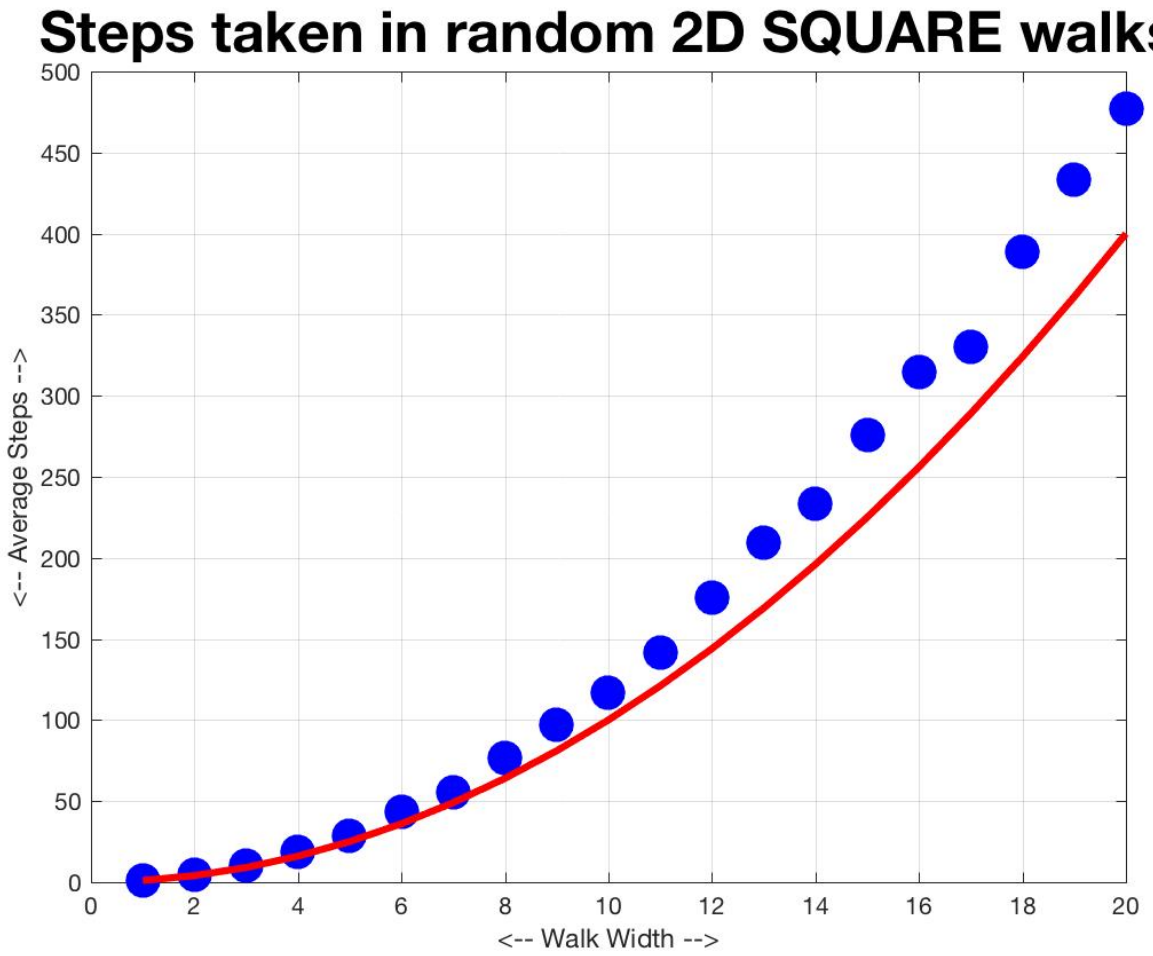


Look for Average Behavior

As in the 1D case, the length of a random walk in 2D is unpredictable. But we can seek the average number of steps taken on a board of dimensions [-N,N]x[-N,N].

It would be interesting to see if we find a pattern similar to what we saw in 1D, that the number of steps is related somehow to N^2.

Step Lengths Pull Away From N^2



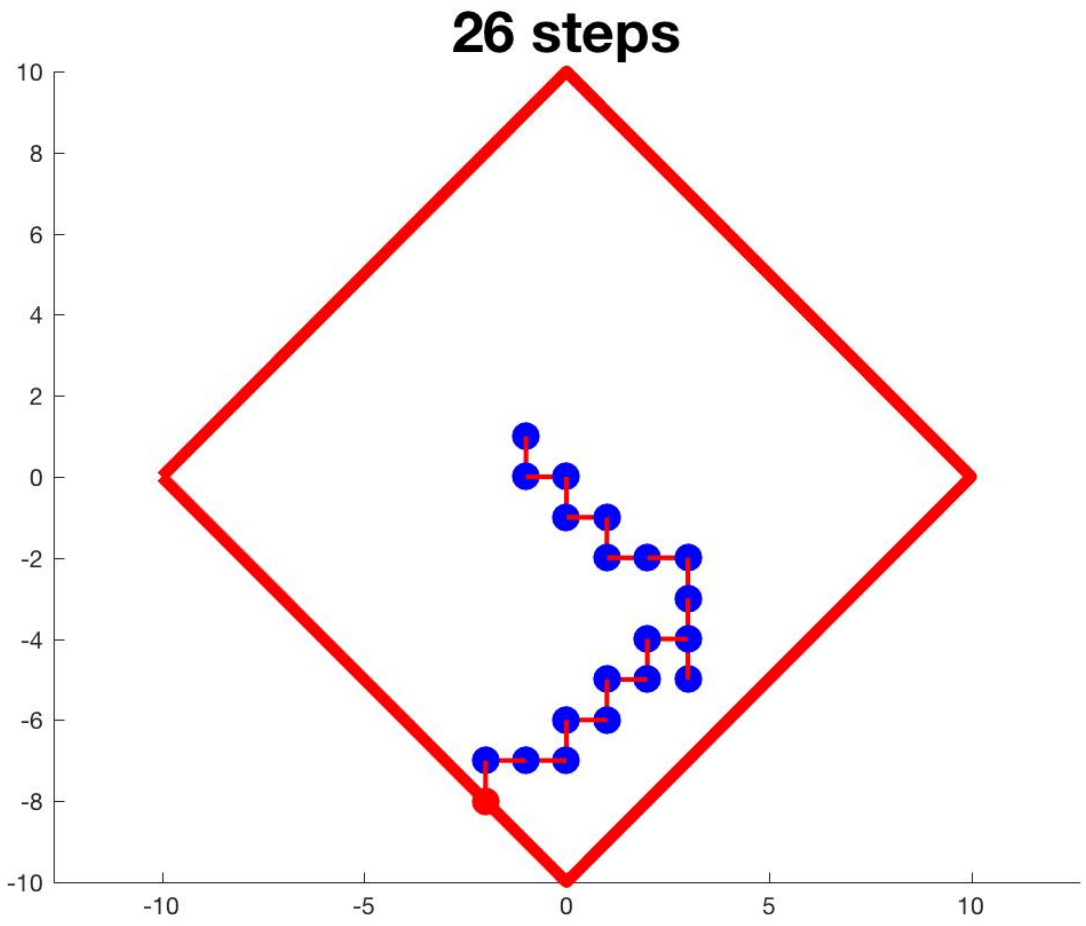
Try a Diamond Shape

The data for the square seems to separate from the N^2 curve.

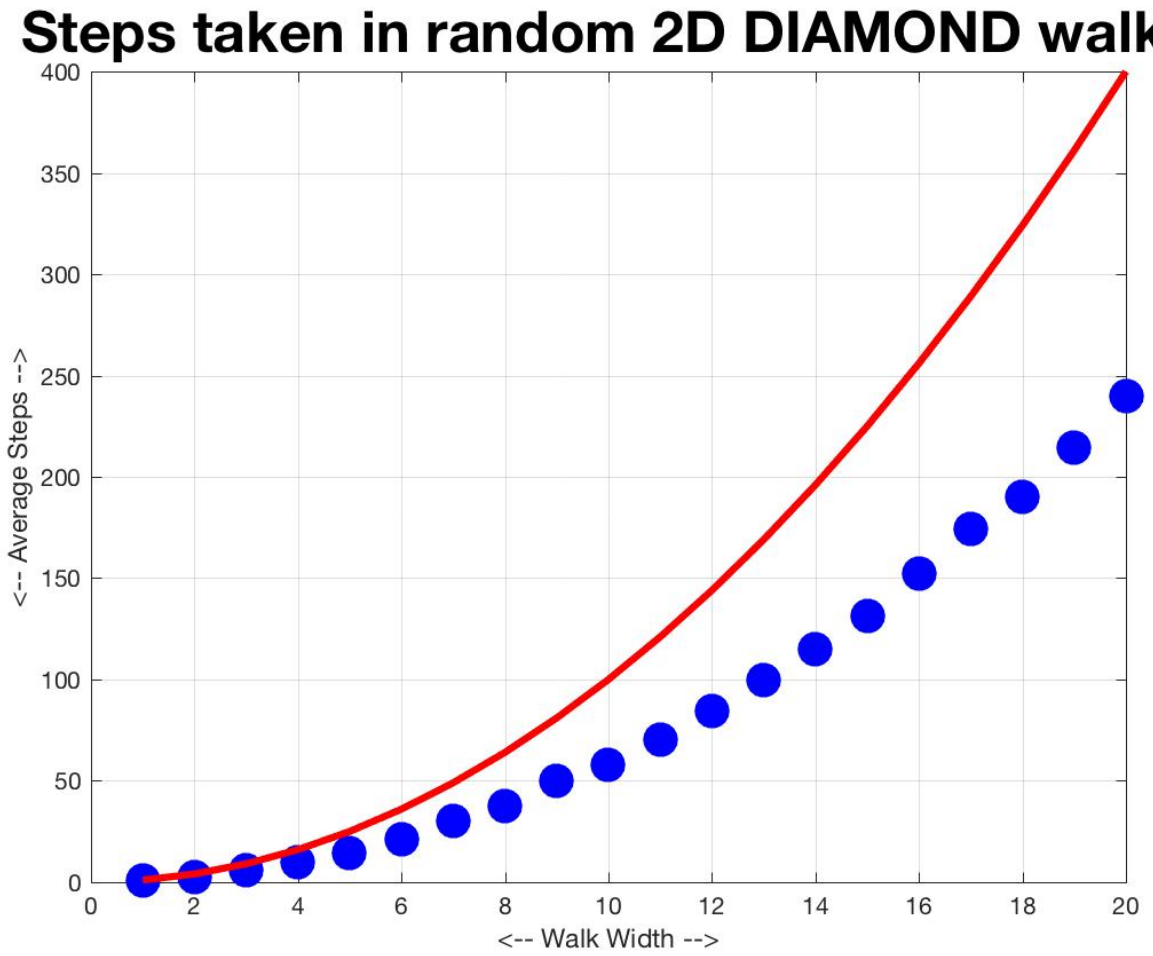
Perhaps we could fix this problem by changing the rules for how we terminate the walk in 2D. Instead of stopping when |X| or |Y| reaches N, we could stop when |X| + |Y| reaches N.

This actually changes the region to a DIAMOND shape.

Try Random Walks in Diamond



Diamond Walk Lengths Go Below N^2



What about a CIRCLE?

A third choice for terminating the walk is to stop when N <= sqrt(X^2+Y^2). This is the same as using a CIRCLE for the boundary.

Notice that each of our three rules is a generalization, for 2D, of the distance function in 1D.

Mathematicians have explored the different geometries associated with these three rules, known as the max norm, the l1 norm, and the Euclidean norm.

Length Norms in 2D

If P=(x,y) is a point in 2D, there are several different norms we can use to measure the length of P, that is, the distance of P from the origin. The one we think is natural is the Euclidean norm.

Euclidean or "2" norm:

||P|| = sqrt (x^2 + y^2) = norm (P, 2) = norm (P);

Max or "infinity" norm:

||P|| = max (abs(x), abs(y)) = norm (P, Inf);

Manhattan or "1" norm:

||P|| = abs(x) + abs(y) = norm (P, 1);

Distance Norms in 2D

If P1=(x1,y1) and P2 = (x2,y2) are points in 2D, then the same norms can be used to measure the distance between P1 and P2.

Euclidean or "2" norm:

||P1-P2|| = sqrt ((x1-x2)^2 + (y1-y2)^2)

= norm (P1-P2, 2) = norm (P1-P2);

Max or "infinity" norm:

||P1-P2|| = max (abs(x1-x2), abs(y1-y2))

= norm (P1-P2, Inf);

Manhattan or "1" norm:

||P1-P2|| = abs(x1-x2) + abs(y1-y2)

= norm (P1-P2, 1);

Examples

P = [1, 2];

norm(P) = sqrt(1+4) = 2.2361;

norm(P,1) = 1 + 2 = 3;

norm(P,Inf) = max(1,2) = 2;

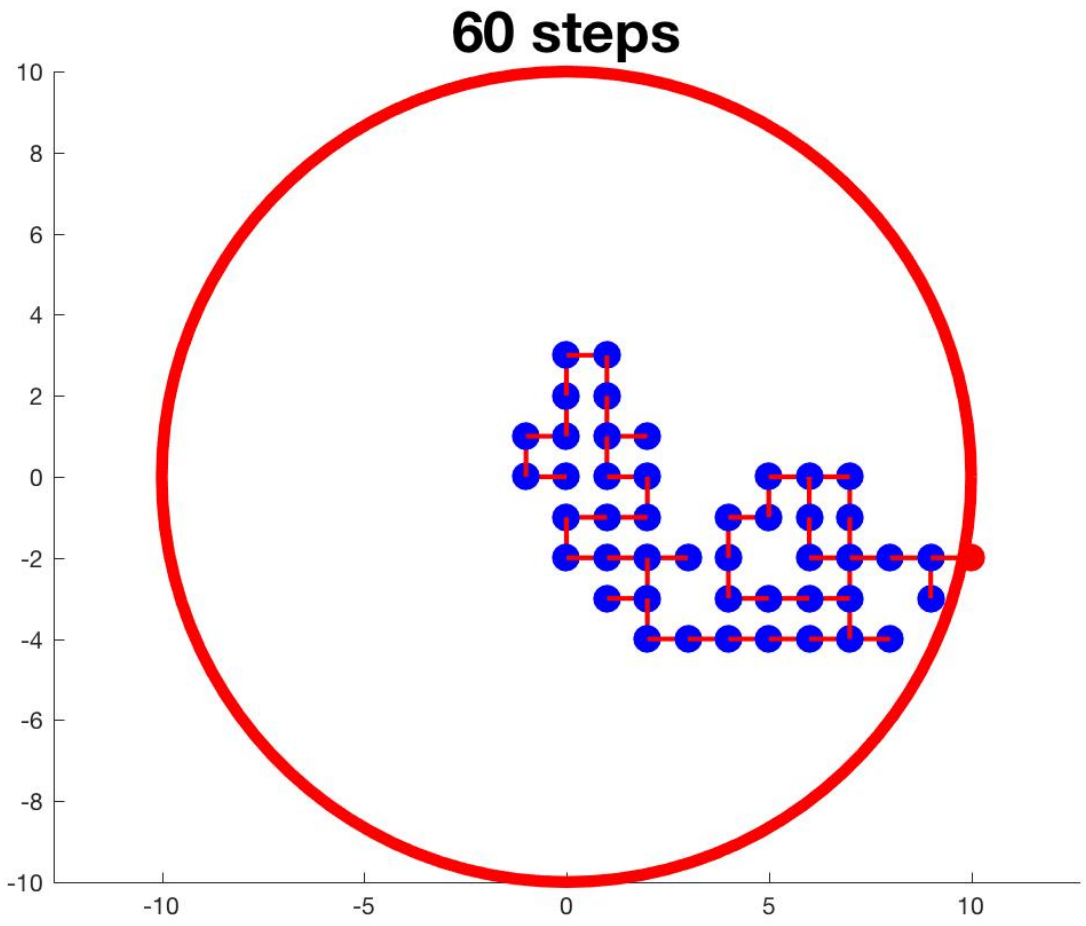
P1 = [1, 2], P2 = [3, 7];

norm(P1-P2) = sqrt(2^2+5^2) = 5.3852

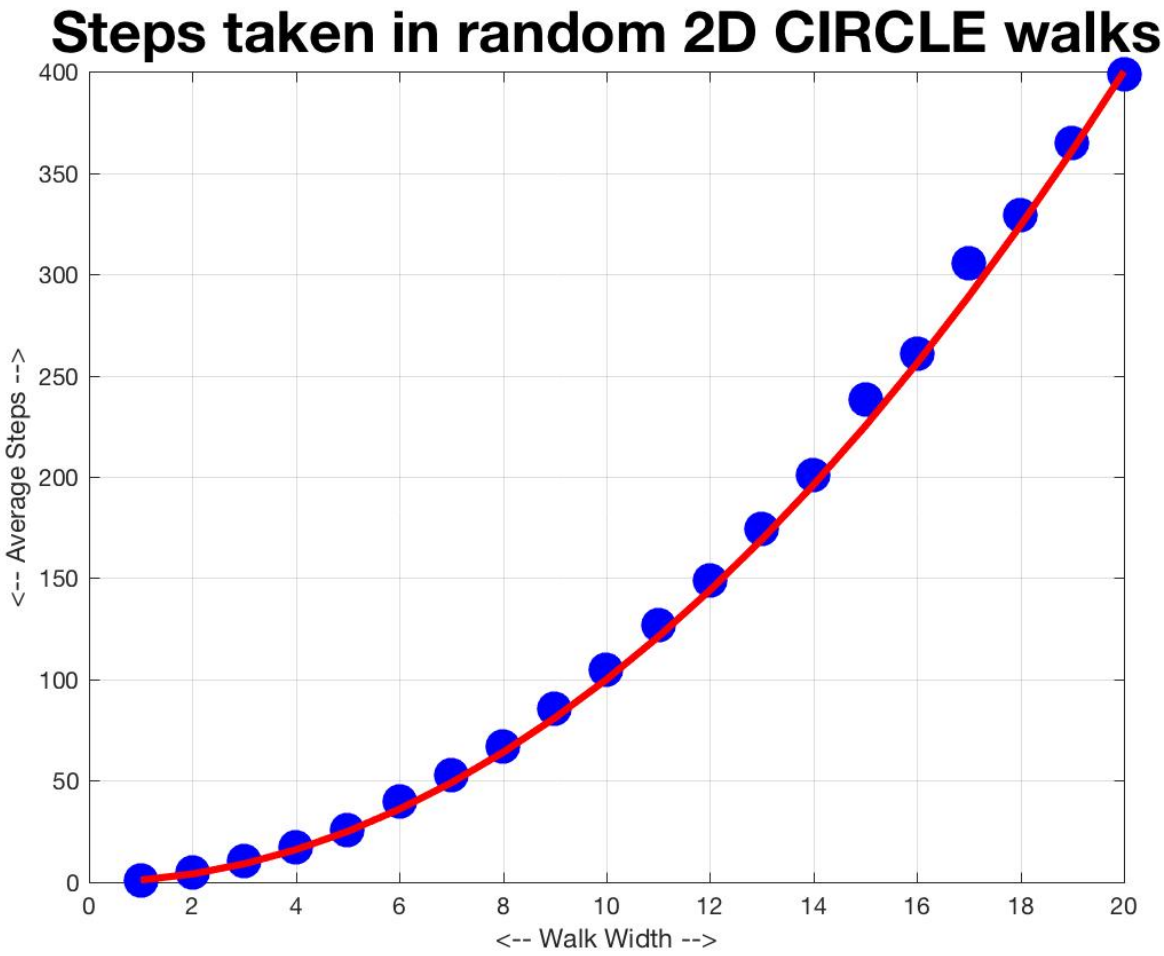
norm(P1-P2,1) = 2+5 = 7;

norm(P1-P2,Inf) = 5;

Try Walks in Circle



Walk Lengths Closely Match N^2



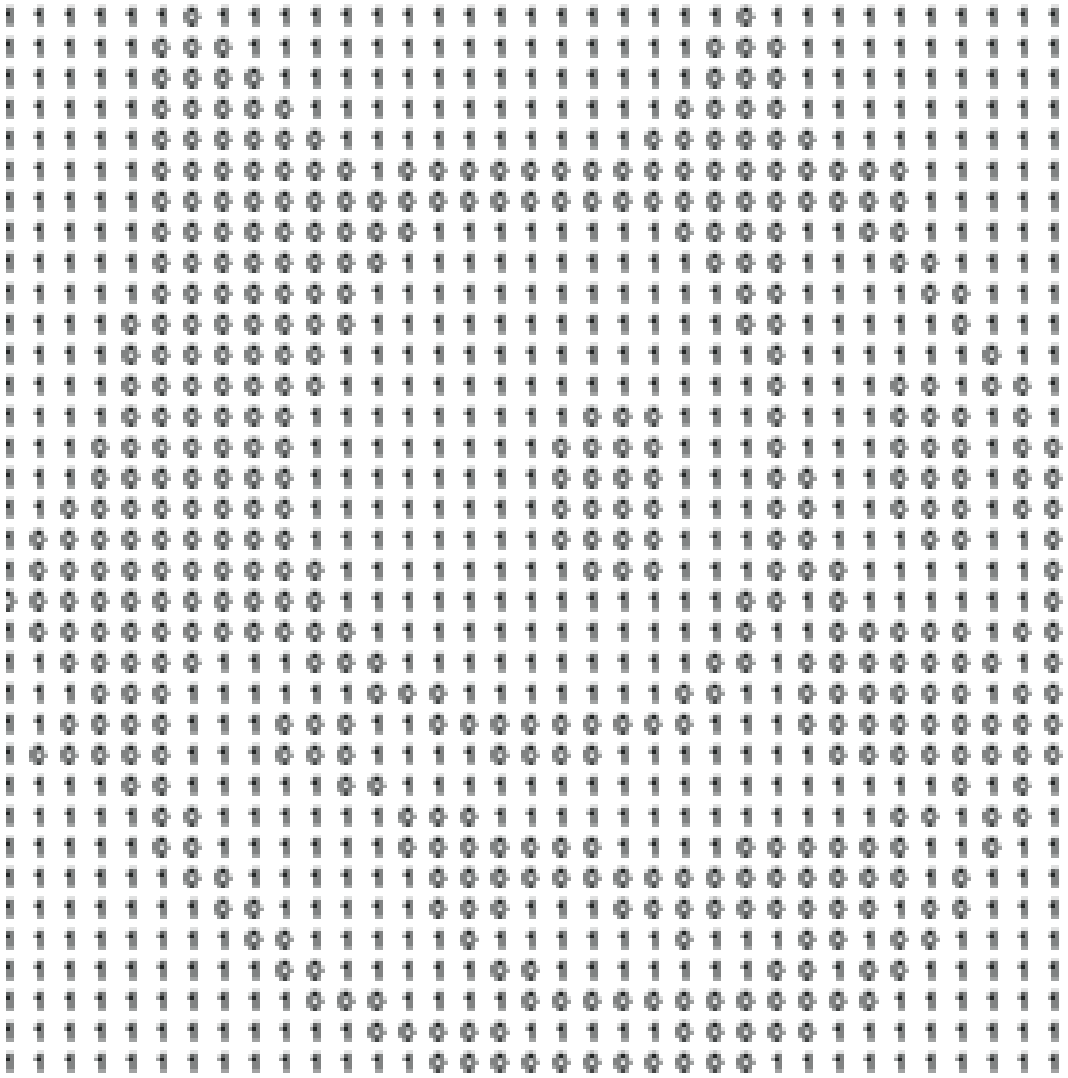
Conclusions

By estimating the average walk lengths in 1D, and then trying to transfer the results to 2D, we have "discovered" that the Euclidean norm seems to be the right way to measure distance for this problem.

If we moved to a 3D random walk, our region would be a sphere, instead of a rectangular box or diamond.

This actually makes sense for the physical problems modeled by random walks, such as Brownian motion, diffusion of ink in water, and the way that heat is transmitted in a sheet of metal.

Working With Data in a 2D Matrix



2D Arrays in MATLAB

In MATLAB, an array is a kind of variable that has a shape (M rows and N columns), which allows it to store M*N values.

To access a particular value in the array, we specify a row I, and a column J, where 1 <= I <= M and 1 <= J <= N.

If the array is named "A", then we access an entry by an expression like "A(I,J)".

It is possible to index matrix entries with a single number K, which runs from 1 to M*N. K starts at A(1,1), runs down the first column, then the second, and so on.

Note that many MATLAB programmers use a capital letter to represent an array, leaving lower case letters for scalars (numbers) and vectors (row vectors or column vectors).

Matrix Functions

```
[ m, n ] = size ( A );

A = zeros ( m, n );
A = ones ( m, n );
A = rand ( m, n );
A = randn ( m, n );
A = randi ( [ a, b ], m, n );

A = [ 11, 12, 13; 21, 22, 23; 31, 32, 33; 41, 42, 43 ];

A = [ 11, 12, 13;
      21, 22, 23;
      31, 32, 33;
      41, 42, 43 ];
```

Matrix Functions

The transpose of a matrix is created using the same apostrophe or ' sign that we used to convert a row vector to a column vector:

```
A = [ 11, 12, 13;
      21, 22, 23;
      31, 32, 33;
      41, 42, 43 ];    a 4x3 matrix

A' = [ 11, 21, 31, 41;
       12, 22, 32, 42;
       13, 23, 33, 43 ]; a 3x4 matrix
```

Some surprises

max(), mean(), min(), std() and sum() return the maximum, mean, minimum, standard deviation and sum of each column of the matrix.

```
A = [ 11, 12, 13;
      21, 22, 23;
      31, 32, 33;
      41, 42, 43 ];
```

max(A) is [41, 42, 43];

min(A) is [11, 12, 13];

sum(A) is [104, 108, 112];

max(max(A)) is 43;

sum(sum(A)) is 324;

How would you get the maximum of each ROW of A?

Add a row or column

```
A = [ 11, 12;
      21, 22];

A2 = [ A; [ 31, 32 ] ]
   is [ 11, 12;
        21, 22;
        31, 32 ];

A3 = [ A, [13;23] ];
   is [ 11, 12, 13;
        21, 22, 23 ];
```

Colon Operations

```
A = [ 11, 12, 12;
      21, 22, 23;
      31, 32, 33 ];
```

```
A(2,3) is 23;                  <- a number
A(2,1:2) is [ 21, 22 ];        <- a row vector
A(2:3,3) is [23; 33];          <- column vector!
A(2:3,1:2) is [ 21, 22;        <- a submatrix
                31, 32 ];
```

Moving Data with Colon Operators

We can copy a row of *A* into a row vector:

```
x = A(2,1:3);
```

We can copy a column vector into a column of *A*:

```
y = [101;201;301];
A(1:3,1) = y;
```

We can overwrite the first two rows of *A* with the last two rows:

```
A(1:2,1:3) = A(2:3,1:3);
```

Logical Operators

Just as we did with vectors, we can specify a logical condition, and make a copy of a matrix with a 1 where the condition is true, 0 elsewhere

```
A = [ 11, 12, 12;
      21, 22, 23;
      31, 32, 33 ];

I = ( mod ( A, 3 ) == 0 )
I is [ 0, 1, 0;
       1, 0, 0;
       0, 0, 1 ];
```

The FIND function

The find() function can be used on a matrix, and its results can be used to index the matrix. However, instead of (I,J) indexing, find() returns a single index K that runs through the matrix a column at a time.

```
A = [ 11, 12, 12;
      21, 22, 23;
      31, 32, 33 ];
```

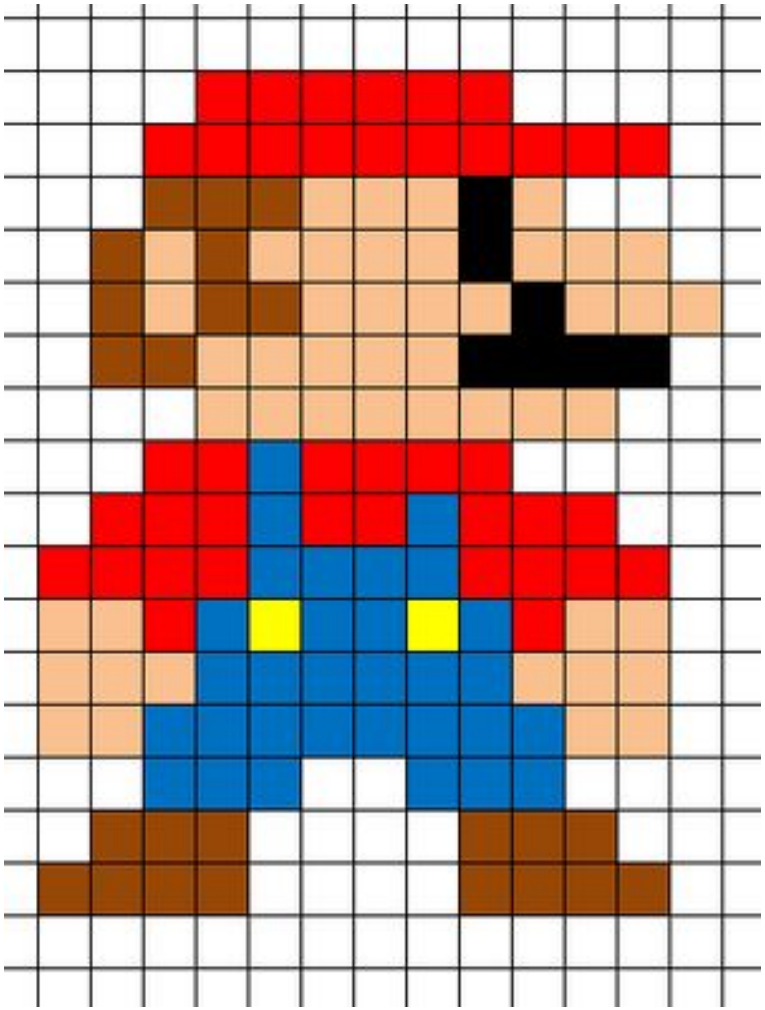
```
Numbering: 1  4  7
           2  5  8
           3  6  9
```

```
k = find ( mod ( A, 10 ) == 2 )
```

k is [4, 5, 6]

A(k) is [12, 22, 32].

Storing MARIO in a 2D Matrix



Storing Mario

The image of Mario can be stored in a matrix. It is already marked up into 16 rows and 13 columns. There are 7 colors in the picture. We can use values 0 through 6 to reference them:

0: white = [1.0, 1.0, 1.0]
1: black = [0.0, 0.0, 0.0]
2: red = [1.0, 0.0, 0.0]
3: blue = [0.0, 0.0, 1.0]
4: yellow = [1.0, 1.0, 0.0]
5: beige = [1.0, 0.8, 0.6]
6: brown = [0.8, 0.4, 0.0]

Find [R,G,B] codes for colors like "beige" at
https://www.w3schools.com/colors/colors_picker.asp

Probably need to divide each value by 255.

Define the Mario Matrix

```
MARIO = [ ...
 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0;
 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0;
 0, 0, 6, 6, 6, 5, 5, 5, 1, 5, 0, 0, 0;
 0, 6, 5, 6, 5, 5, 5, 5, 1, 5, 5, 5, 0;
 0, 6, 5, 6, 6, 5, 5, 5, 5, 1, 5, 5, 5;
 0, 6, 6, 5, 5, 5, 5, 5, 1, 1, 1, 1, 0;
 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 0, 0;
 0, 0, 2, 2, 3, 2, 2, 2, 2, 0, 0, 0, 0;
 0, 2, 2, 2, 3, 2, 2, 3, 2, 2, 2, 0, 0;
 2, 2, 2, 2, 3, 3, 3, 3, 2, 2, 2, 2, 0;
 5, 5, 2, 3, 4, 3, 3, 4, 3, 2, 5, 5, 0;
 5, 5, 5, 3, 3, 3, 3, 3, 3, 5, 5, 5, 0;
 5, 5, 3, 3, 3, 3, 3, 3, 3, 3, 5, 5, 0;
 0, 0, 3, 3, 3, 0, 0, 3, 3, 3, 0, 0, 0;
 0, 6, 6, 6, 0, 0, 0, 0, 6, 6, 6, 0, 0;
 6, 6, 6, 6, 0, 0, 0, 0, 6, 6, 6, 6, 0];
```

We need to draw colored boxes

To recreate the image of Mario, we just have to make a plot of 16 rows and 13 columns of rectangles, each one filled with the appropriate color. There's a MATLAB function to do this:

```
fill ( x, y, color )
```

where x and y are the corners of the shape we are filling with color.

Draw cells one by one

```
for i = 1 : m
  for j = 1 : n

    k = MARIO(i,j);  % <- Get the color index

    if ( k == 0 )
      color = [ 1.0, 1.0, 1.0 ];
    elseif ( k == 1 )
      color = [ 0.0, 0.0, 0.0 ];
    elseif ( k == 2 )
      color = [ 1.0, 0.0, 0.0 ];
    elseif ( k == 3 )
      color = [ 0.0, 0.0, 1.0 ];
    elseif ( k == 4 )
      color = [ 1.0, 1.0, 0.0 ];
    elseif ( k == 5 )
      color = [ 1.0, 0.8, 0.6 ];
    elseif ( k == 6 )
      color = [ 0.8, 0.4, 0.0 ];
    end

    FILL CELL(I,J)  WITH COLOR BY CALLING FILL IN THE RIGHT WAY!

  end
end
```

(X,Y) coordinates of cell (I,J)?

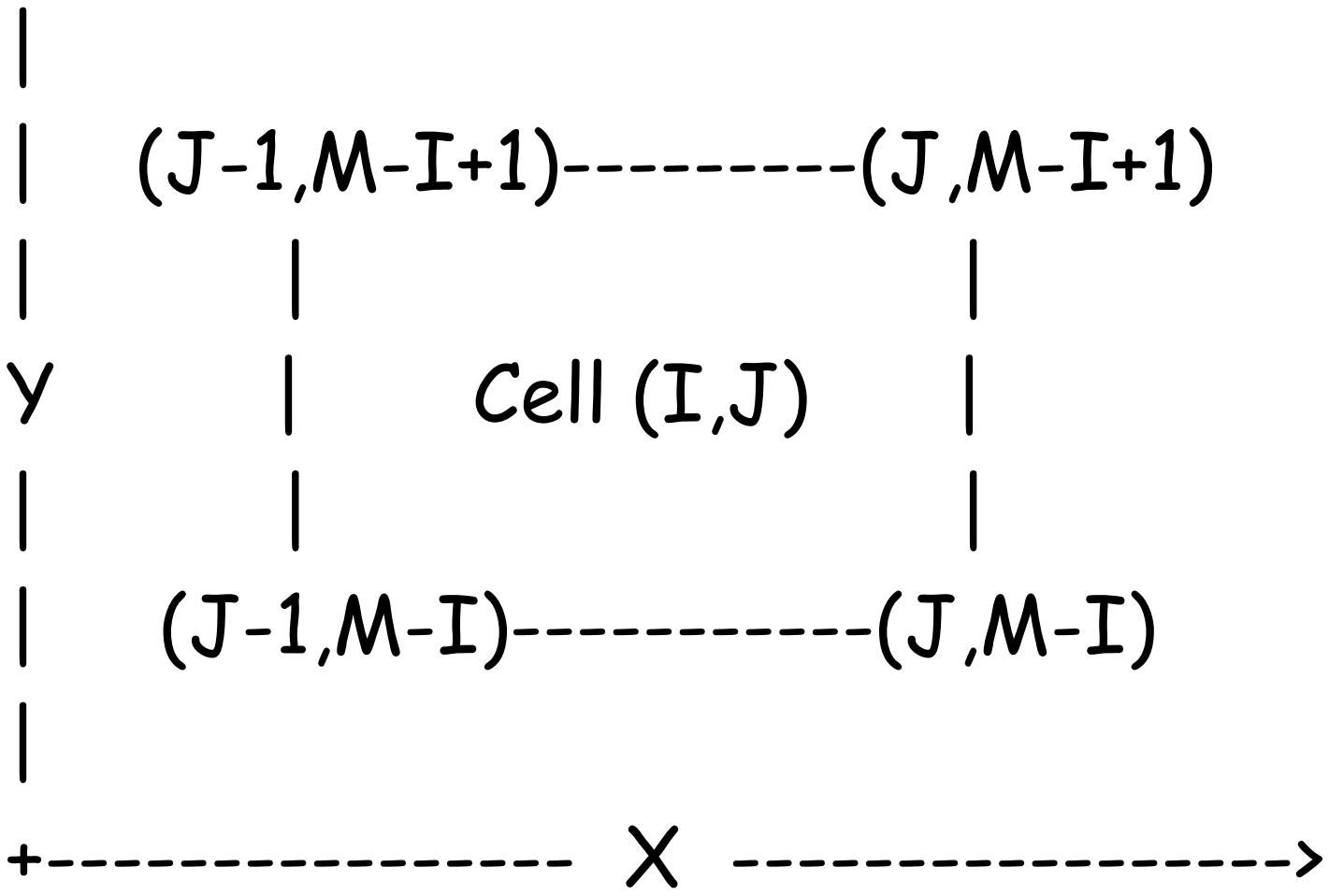
Note that the (X,Y) coordinate system used by fill() is very different from the (I,J) coordinate system used in the matrix MARIO!

As I gets bigger, we move down the matrix. As X gets bigger, we move right, so I is more like the Y coordinate, but moving down when Y moves up!

The J coordinate moves right, like X does.

So we have to remember to plot entry (I,J) we used (X,Y) coordinates like (J,M-I).

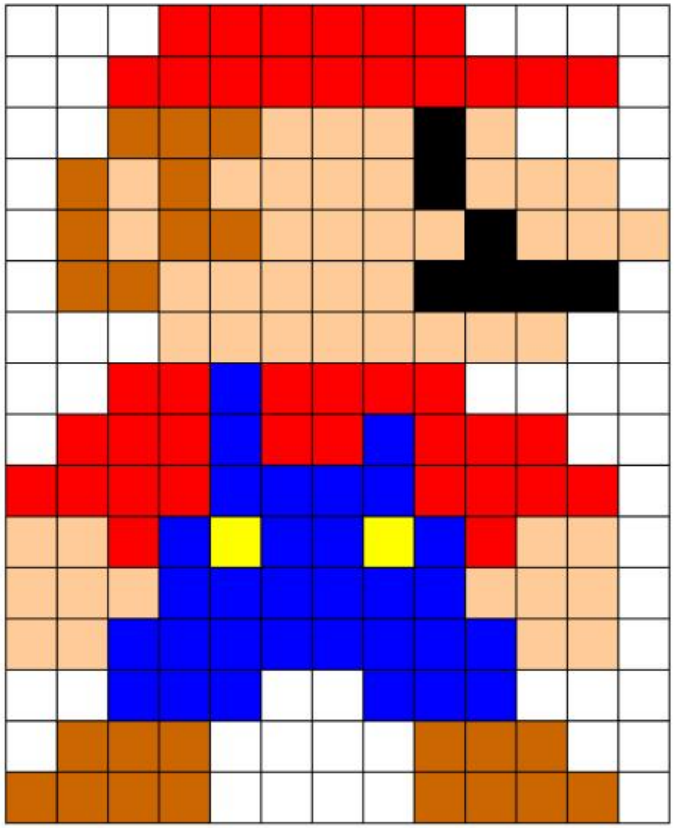
The box forming cell (I,J)



How to call FILL()

```
for i = 1 : m
  for j = 1 : n

    k = MARIO(i,j);

    if ( k == 0 )
      color = [ 1.0, 1.0, 1.0 ];
      MORE COLOR STUFF...
   end

    a = j - 1;
    b = j;
    c = m - i + 1;
    d = m - i;

    fill ( [ a, b, b, a ], [ c, c, d, d ], color );

  end
end
```

Our Version of Mario



Get Credit for One Homework Problem

We will solve the tasks on the following page.

Refer to the program "mario.m" available in today's Canvas directory.

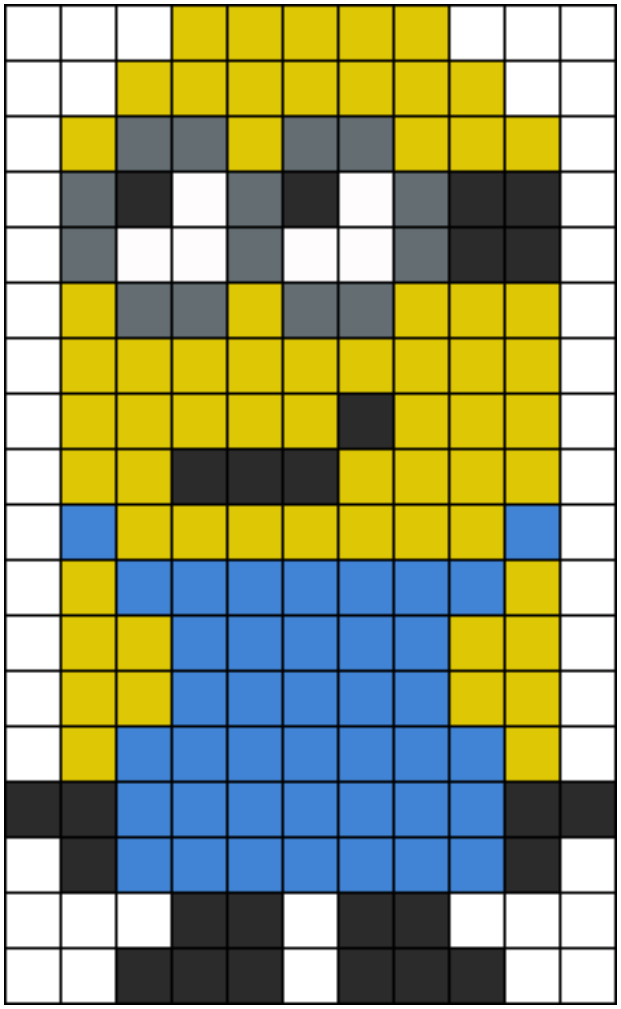
Suggest a change to the program that will carry out the desired task.

If I choose your solution, you can skip ONE problem in homework #7 OR homework #8, and get full credit for it. (But remind me in your homework submission!)

Matrix Practice

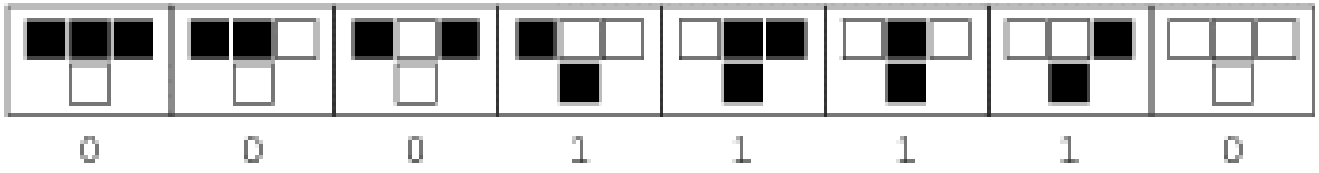
1: Change Mario's eye color to GRAY.
2: Change Mario's pants to ORANGE.
3: Give Mario a PURPLE belt.
4: Change RED to MAROON.
5: Change just one button to CYAN.
6: Put Mario behind BLACK bars.
7: Change the background to PALE BLUE.
8: Place Mario in 4 rows of GREEN water.
9: Can you flip Mario so he looks left?

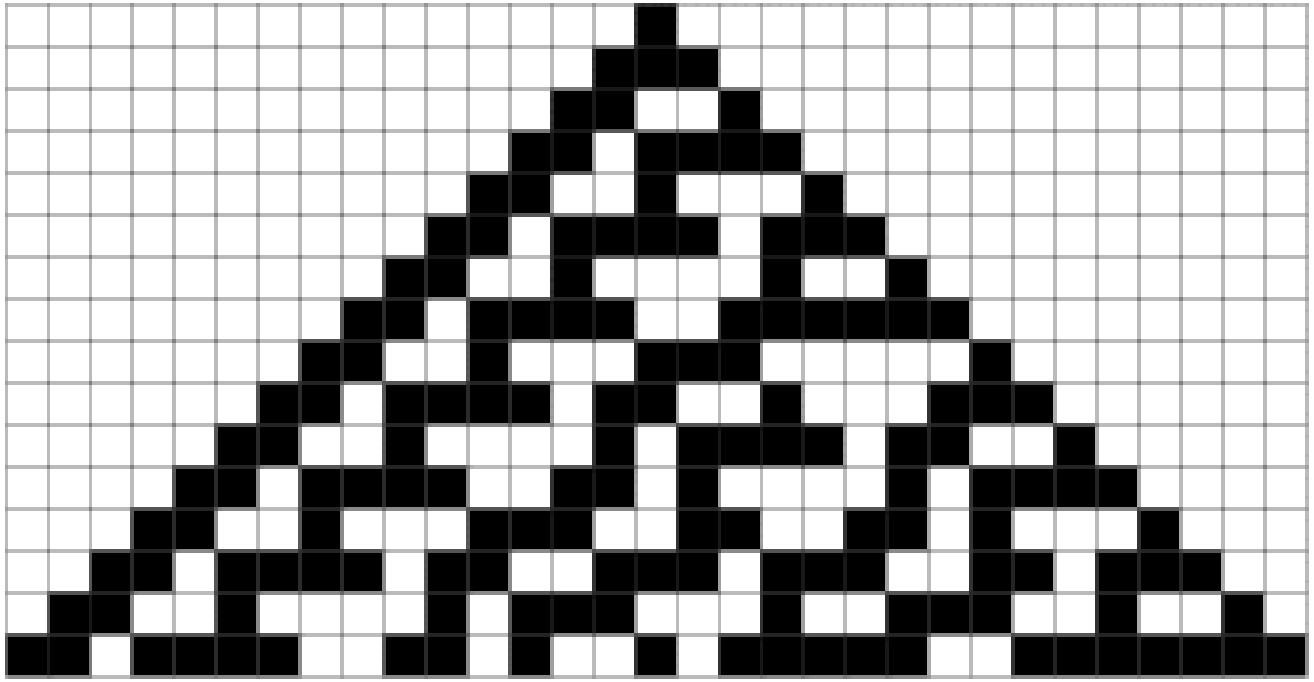
Another Challenge for You



A Cellular Automaton

rule 30





Cellular Automatons

A cellular automaton is a grid of cells, each of which has a "state" (which might be a number or color.)

On each time step, the cellular automaton updates the state of all the cells, according to a set of rules based on the state of neighbor cells.

In the simplest case, the grid is a simple row vector, the state of each cell is "black" or "white", and each cell makes its decision based on its current state, and that of its left and right neighbors.

Example State Rule

0: 0,0,0 -> 0
1: 0,0,1 -> 1
2: 0,1,0 -> 1
3: 0,1,1 -> 1
4: 1,0,0 -> 1
5: 1,0,1 -> 0
6: 1,1,0 -> 0
7: 1,1,1 -> 0

0,0,0,1,0,0,0 -> 0 (don't change)
0,0,0,1,0,0,0 -> 0
0,0,0,1,0,0,0 -> 1
0,0,0,1,0,0,0 -> 1
0,0,0,1,0,0,0 -> 1
0,0,0,1,0,0,0 -> 0
0,0,0,1,0,0,0 -> 0 (don't change)

So new grid is

0,0,1,1,1,0,0

Work Out 10 Steps

0: 0,0,0 -> 0
1: 0,0,1 -> 1
2: 0,1,0 -> 1
3: 0,1,1 -> 1
4: 1,0,0 -> 1
5: 1,0,1 -> 0
6: 1,1,0 -> 0
7: 1,1,1 -> 0

0: 0 0 0 1 0 0 0
1: _,_,_,_,_,_,_
2: _,_,_,_,_,_,_
3: _,_,_,_,_,_,_
4: _,_,_,_,_,_,_
5: _,_,_,_,_,_,_
6: _,_,_,_,_,_,_
7: _,_,_,_,_,_,_
8: _,_,_,_,_,_,_
9: _,_,_,_,_,_,_
10: _,_,_,_,_,_,_

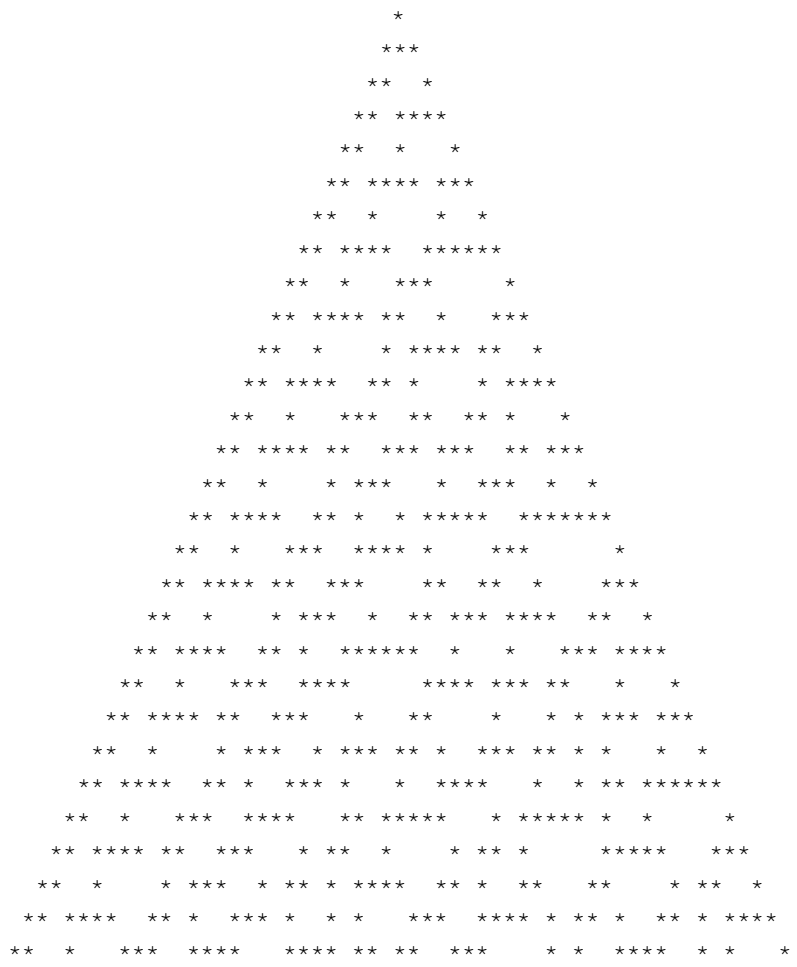
Cellular Automaton: Advance One Step

```
function row_next = ca_step ( row )

  n = length ( row );

  row_next = zeros ( 1, n );

  for i = 2 : n - 1

    if ( ( row(i-1) == 0 && row(i) == 0 && row(i+1) == 1 ) || ...
         ( row(i-1) == 0 && row(i) == 1 && row(i+1) == 0 ) || ...
         ( row(i-1) == 0 && row(i) == 1 && row(i+1) == 1 ) || ...
         ( row(i-1) == 1 && row(i) == 0 && row(i+1) == 0 ) )
      row_next(i) = 1;
    else
      row_next(i) = 0;
    end

  end

  return
end
```

Print 0's and 1's

```
00000000000000000000000000000000000000010000000000000000000000000000000000000000
00000000000000000000000000000000000001110000000000000000000000000000000000000000
00000000000000000000000000000000000011001000000000000000000000000000000000000000
00000000000000000000000000000000000110111100000000000000000000000000000000000000
00000000000000000000000000000000001100100010000000000000000000000000000000000000
00000000000000000000000000000000011011110111000000000000000000000000000000000000
00000000000000000000000000000000110010000100100000000000000000000000000000000000
00000000000000000000000000000000110111100111111000000000000000000000000000000000
00000000000000000000000000000001100100001110000010000000000000000000000000000000
00000000000000000000000000000011011110110010001110000000000000000000000000000000
00000000000000000000000000000110010000101111011001000000000000000000000000000000
00000000000000000000000000001101111011001110111001101110000000000000000000000000
00000000000000000000000000011001000010111000100111001001000000000000000000000000
00000000000000000000000001101111011001110100101111100111111100000000000000000000
00000000000000000000000110010000111001111010000111000000100000000000000000000000
00000000000000000000001101111011001110000110011001000011100000000000000000000000
00000000000000000000011001000010111001001101110111001100100000000000000000000000
00000000000000000000110111100110100111111001000100011101111000000000000000000000
00000000000000000001100100011100111100001111011101100010001000000000000000000000
00000000000000000011011110110011100010001100001000101011101110000000000000000000
00000000000000000110010000101110010111011010011101101010001001000000000000000000
00000000000000001101111001101001110100010011110001001011011111000000000000000000
00000000000000011001000111001111000110111110001011111010010000100000000000000000
```

Print * for 1, Blank for 0



Store Data in Array and Plot

To make a nicer plot, we can create an array *CA*, which contains the last M rows of the output.

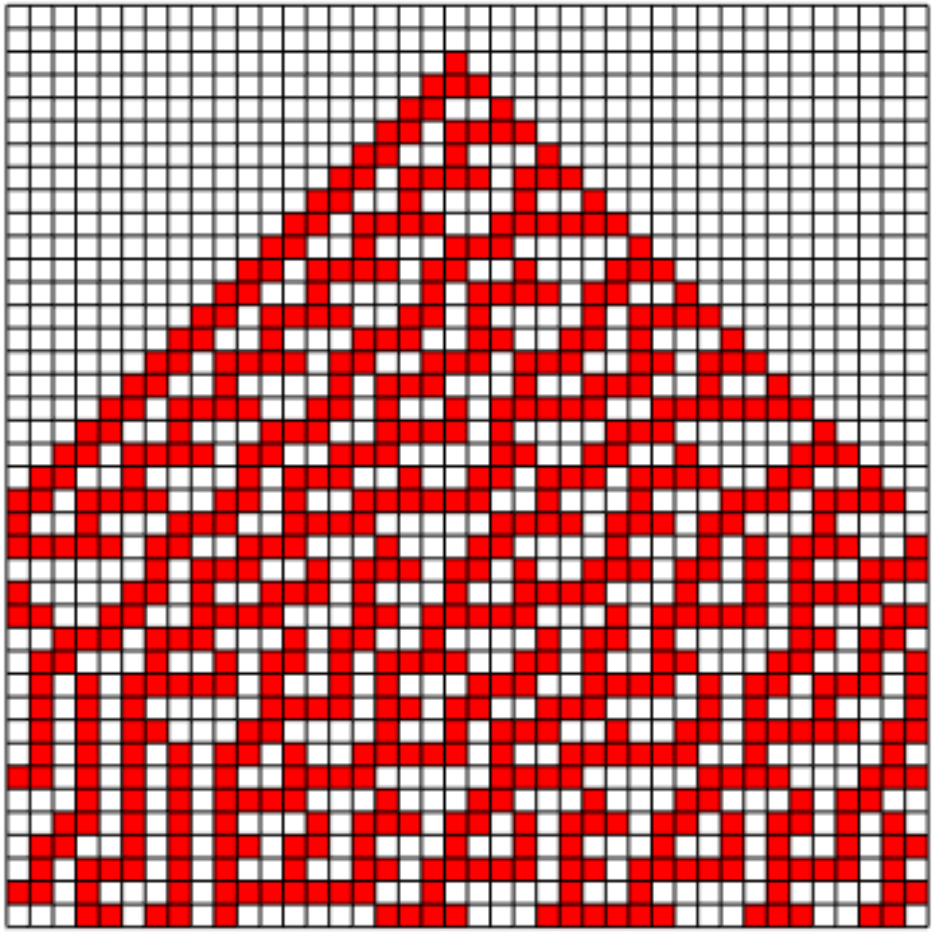
In order to take the NEXT step, we need to copy rows 2 through M of *CA* into rows 1 through M-1, and then compute a new row M.

```
CA(1:M-1,1:N) = CA(2:M,1:N);
CA(M,1:N) = ca_step ( CA(M-1,1:N) );
```

Setting the Cellular Automaton Matrix

```
m = 40;
n = 40;

for k = 0 : 100

  if ( k == 0 )
    ca = zeros ( m, n );
    ca(m,20) = 1;
  else
    ca(1:m-1,:) = ca(2:m,:);
    ca(m,:) = ca_step ( ca(m-1,:) );
  end

  REST OF LOOP PLOTS BLACK OR WHITE BOXES, SIMILAR TO MARIO

end
```

After 38 steps



Wrap Around Boundary

As it stands, the first and last cells just sit at the value "0" forever.

We could make things more interesting by using a "wrap around" condition. In that case, the cell 1 neighborhood would be:

cell(n) cell(1) cell(2)

and the cell N neighborhood would be

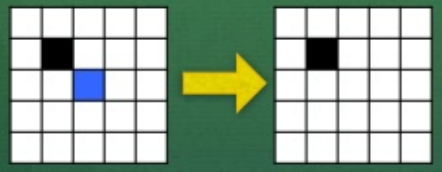
cell(n-1) cell(n) cell(1)

and this means cell(1) and cell(n) can change too.

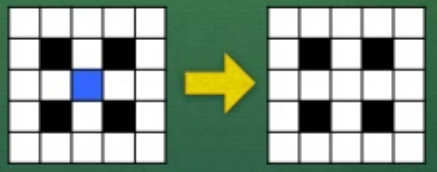
The Game of Life (Next Time)

Basic Rules of Conway's Game of Life

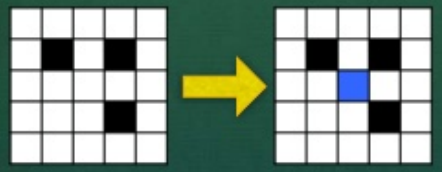
1. Living cells die if they have fewer than 2 neighbors (underpopulation/loneliness)



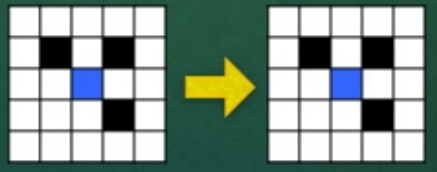
2. Living cells die if they have more than 3 neighbors (overpopulation)



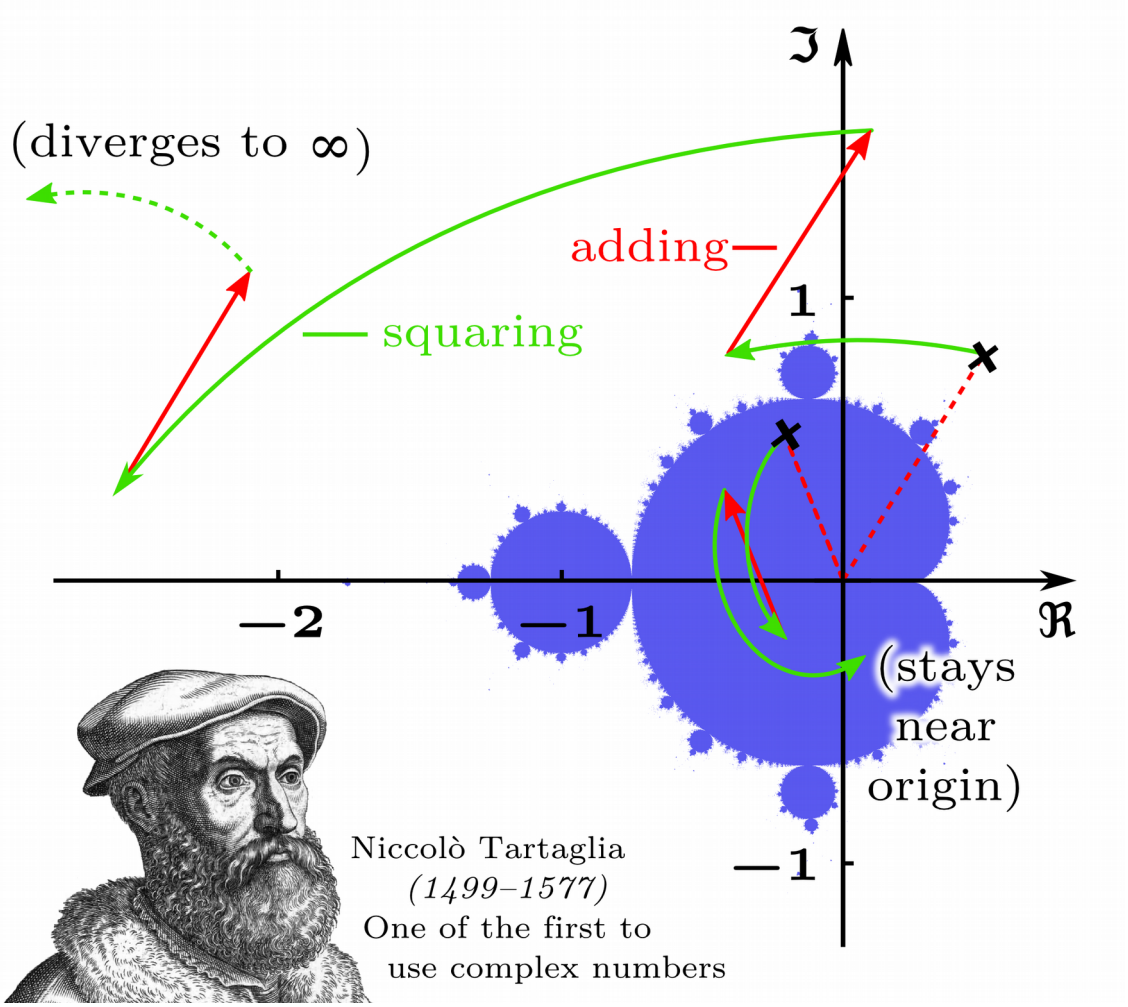
3. Dead cells that have 3 neighbors become alive (reproduction)



4. Otherwise, there is no change (whether cell is alive or dead)



The Mandelbrot Set (Next Time)



Lights Out (Next Time)

Homework #8

HW044: Write a function that uses logical vectors and find() to analyze a vector.

HW045: Write a function that estimates how many times a random walk will visit the square labeled 0.

HW046: Write a function that tracks the frequency of each possible score when tossing N dice M times, and creates a bin plot of the results.

Homework #7 is due Friday night!