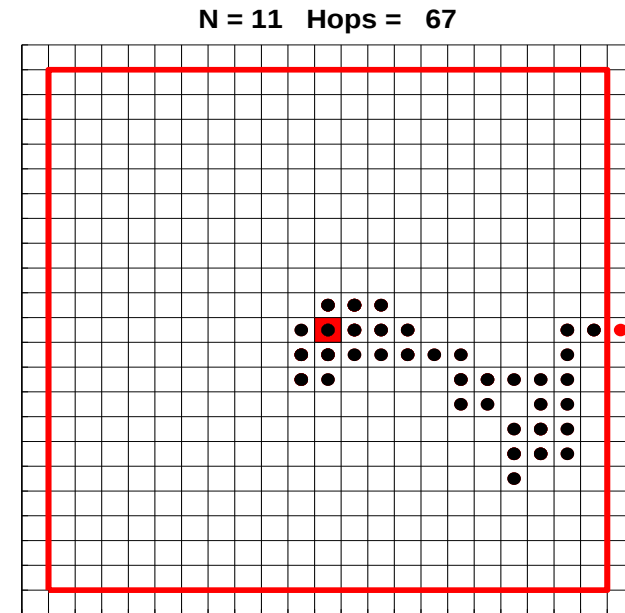
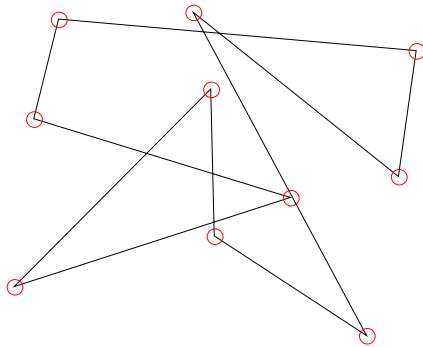


- Previous Lecture:
  - 1-d array—vector
  - Probability and random numbers
  
- Today's Lecture:
  - More examples on vectors and simulation
  
- Announcements:
  - Discussion this week in the computer lab
  - Project 3 due on Mon 10/3

# Simulation

- Imitates real system
- Requires judicious use of random numbers
- Requires many trials
- → opportunity to practice working with vectors!

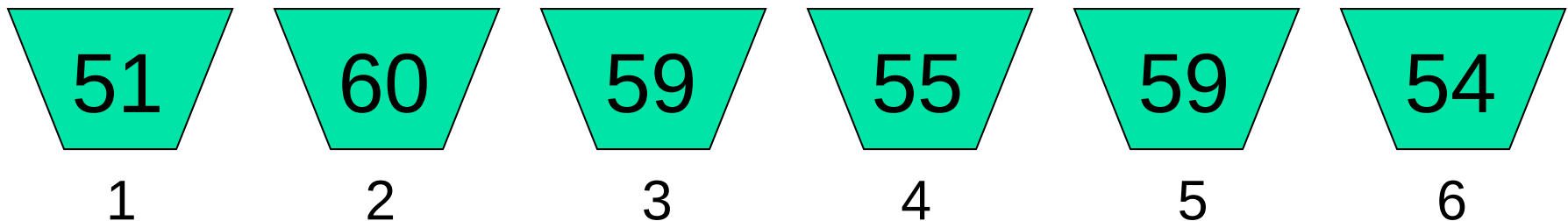
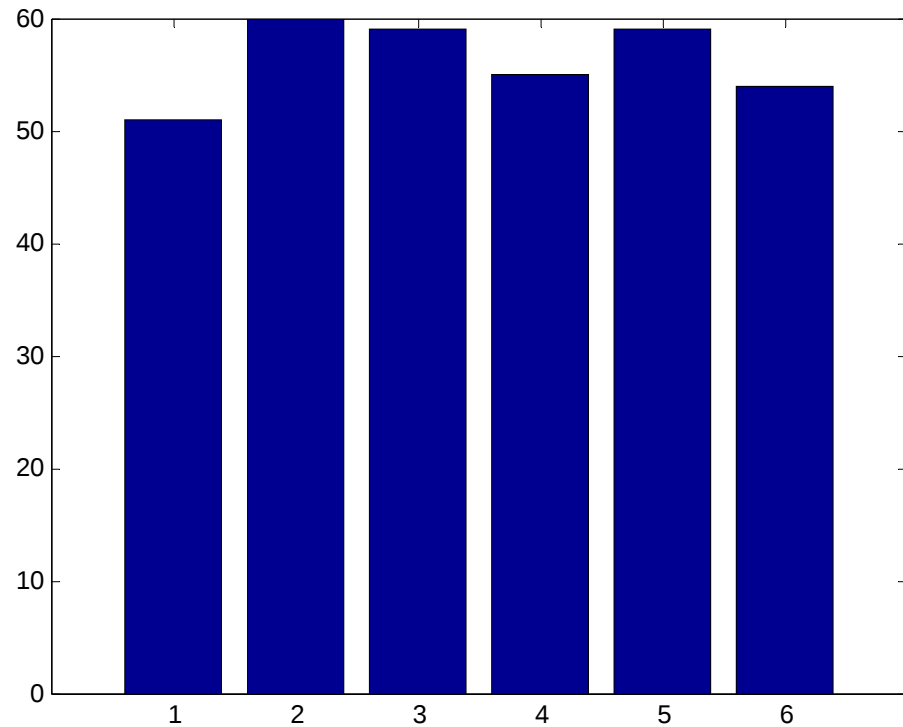


# Simulation result

`bar(1:6, count)`

Bin numbers

Data in bins



Keep tally on repeated rolls of a fair die

***Repeat the following:***

**% roll the die**

**% increment correct “bin”**

```
function count = rollDie(rolls)
```

```
FACES= 6;
```

```
count= zeros(1,FACES);
```

```
% #faces on die
```

|       | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| count | 0 | 0 | 0 | 0 | 0 | 0 |

```
% Count outcomes of rolling a FAIR die
```

```
for k= 1:rolls
```

```
    % Roll kth die
```

```
    % Increment the appropriate bin
```

```
end
```

```
% Show histogram of outcome
```

```
function count = rollDie(rolls)
```

```
FACES= 6;
```

```
count= zeros(1,FACES);
```

```
% #faces on die
```

|       | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| count | 0 | 0 | 0 | 0 | 0 | 0 |

```
% Count outcomes of rolling a FAIR die
```

```
for k= 1:rolls
```

```
    % Roll kth die
```

```
    face= ceil(rand*FACES);
```

```
    % Increment the appropriate bin
```

```
end
```

```
% Show histogram of outcome
```

**% Count outcomes of rolling a FAIR die**

**count= zeros(1,6);**

**for k= 1:100**

**face= ceil(rand\*6);**

**if face==1**

**count(1)= count(1) + 1;**

**elseif face==2**

**count(2)= count(2) + 1;**

**⋮**

**elseif face==5**

**count(5)= count(5) + 1;**

**else**

**count(6)= count(6) + 1;**

**end**

**end**

|       | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| count | 0 | 0 | 0 | 0 | 0 | 0 |

Looks redundant ... is  
it? Is there a more  
concise way?

From **rollDieV1.m**

```
function count = rollDie(rolls)
```

```
FACES= 6;
```

```
count= zeros(1,FACES);
```

```
% #faces on die
```

|       | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| count | 0 | 0 | 0 | 0 | 0 | 0 |

```
% Count outcomes of rolling a FAIR die
```

```
for k= 1:rolls
```

```
    % Roll kth die
```

```
    face= ceil(rand*FACES);
```

```
    % Increment the appropriate bin
```

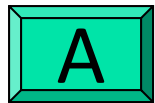
```
    count(face)= count(face) + 1;
```

```
end
```

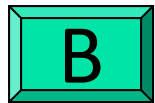
```
% Show histogram of outcome
```

% Simulate the rolling of 2 fair dice

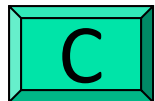
totalOutcome= ???



**ceil(rand\*12)**



**ceil(rand\*11)+1**



**floor(rand\*11)+2**



*2 of the above*



*None of the above*

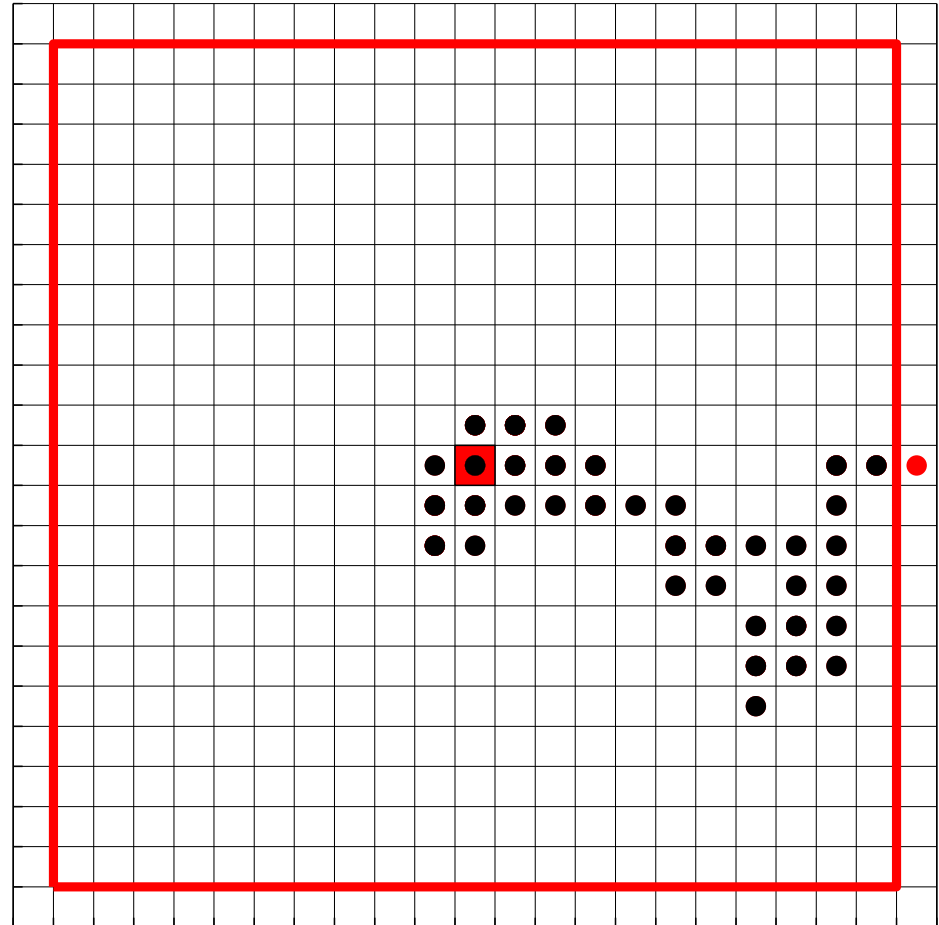
## 2-dimensional random walk

Start in the middle tile,  
(0,0).

For each step,  
randomly choose  
between N,E,S,W and  
then walk one tile.  
Each tile is  $1 \times 1$ .

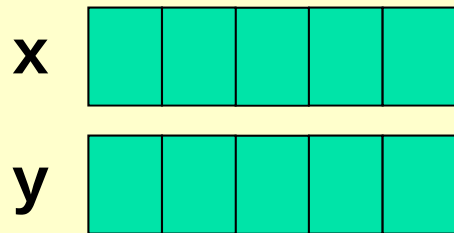
Walk until you reach  
the boundary.

N = 11 Hops = 67



```
function [x, y] = RandomWalk2D(N)
% 2D random walk in 2N-1 by 2N-1 grid.
% Walk randomly from (0,0) to an edge.
% Vectors x,y represent the path.
```

By the end of the function ...



```
function [x, y] = RandomWalk2D(N)
```

```
k=0;   xc=0;   yc=0;
```

```
while not at an edge
```

```
    % Choose random dir, update xc,yc
```

```
    % Record new location in x, y
```

```
end
```

```
function [x, y] = RandomWalk2D(N)
```

```
k=0;   xc=0;   yc=0;
```

```
while   abs(xc)<N && abs(yc)<N
```

```
    % Choose random dir, update xc,yc
```

```
    % Record new location in x, y
```

```
end
```

```
function [x, y] = RandomWalk2D(N)
```

```
k=0;   xc=0;   yc=0;
```

```
while   abs(xc)<N && abs(yc)<N
```

```
    % Choose random dir, update xc,yc
```

```
    % Record new location in x, y
```

```
    k=k+1;   x(k)=xc;   y(k)=yc;
```

```
end
```

```
% Standing at (xc,yc)
% Randomly select a step
r= rand(1);
if r < .25
    yc= yc + 1;    % north
elseif r < .5
    xc= xc + 1;    % east
elseif r < .75
    yc= yc -1;     % south
else
    xc= xc -1;     % west
end
```

[See RandomWalk2D.m](#)

## Another representation for the random step

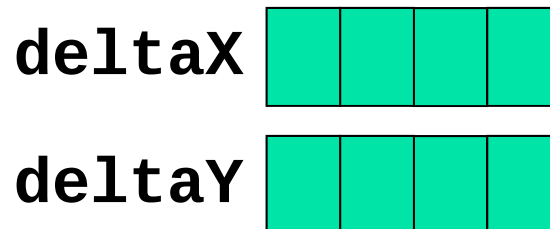
- Observe that each update has the form

$$x_c = x_c + \Delta x$$

$$y_c = y_c + \Delta y$$

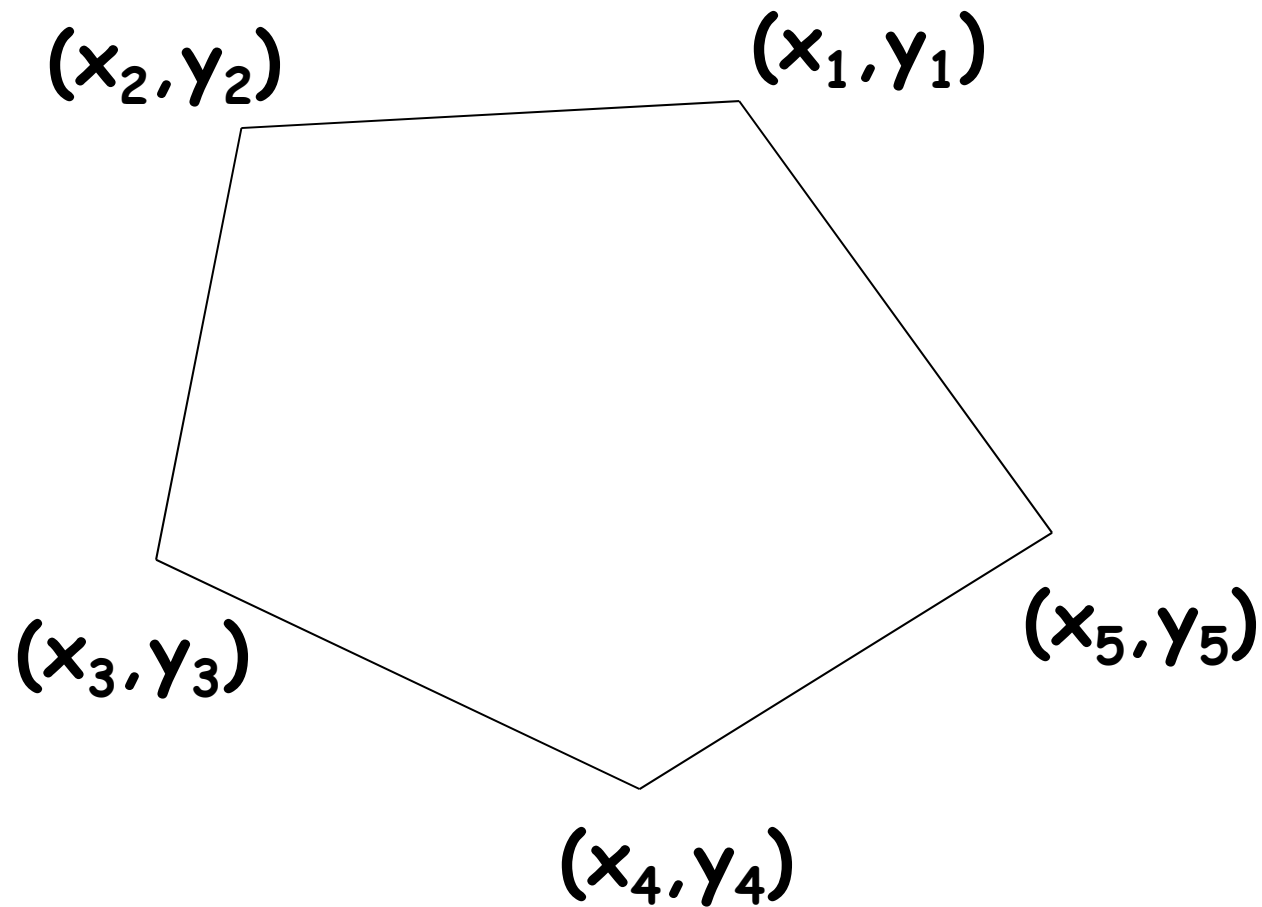
no matter which direction is taken.

- So let's get rid of the if statement!
- Need to create two “change vectors” `deltaX` and `deltaY`

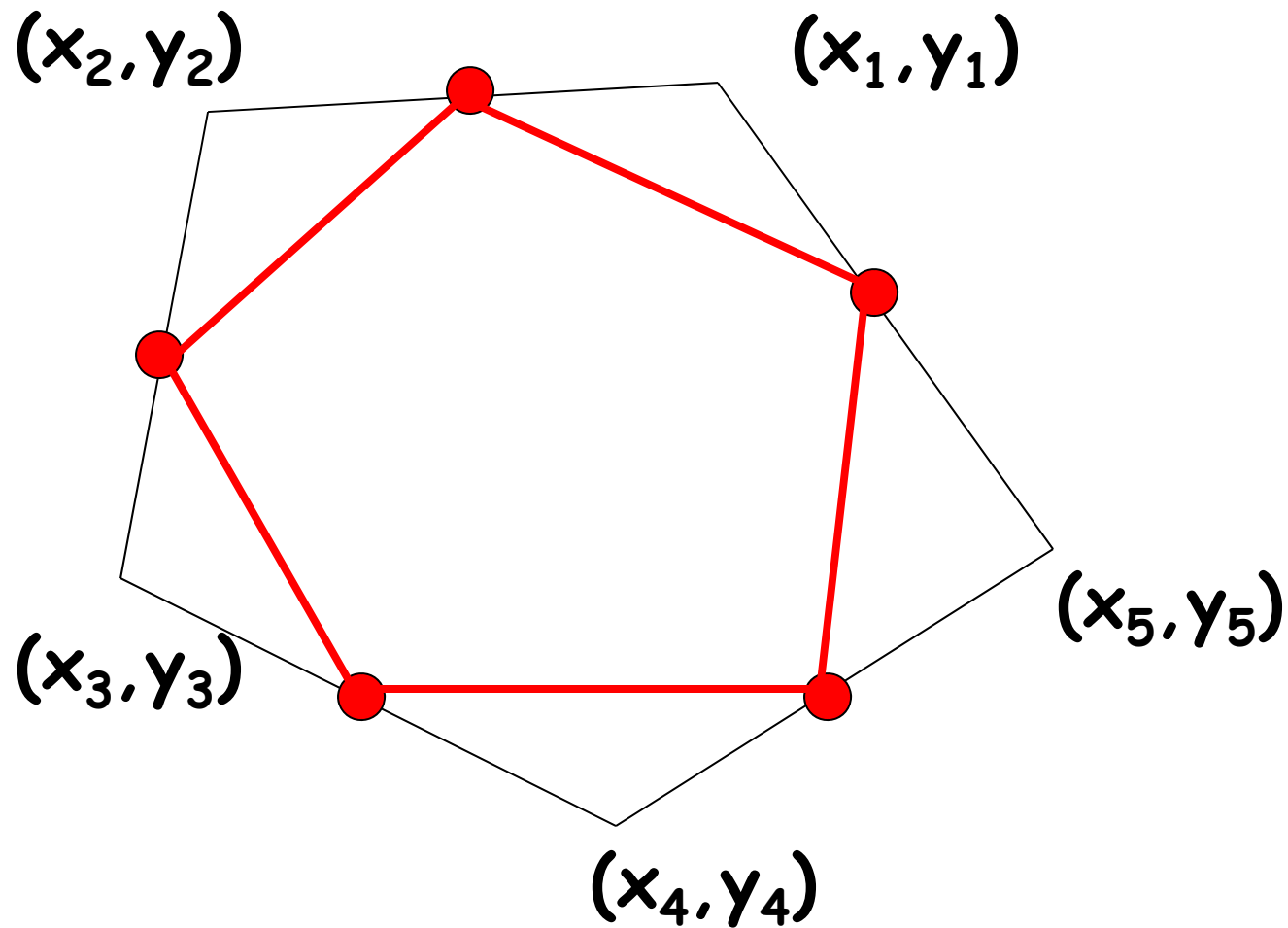


See `RandomWalk2D_v2.m`

## Example: polygon smoothing



## Example: polygon smoothing

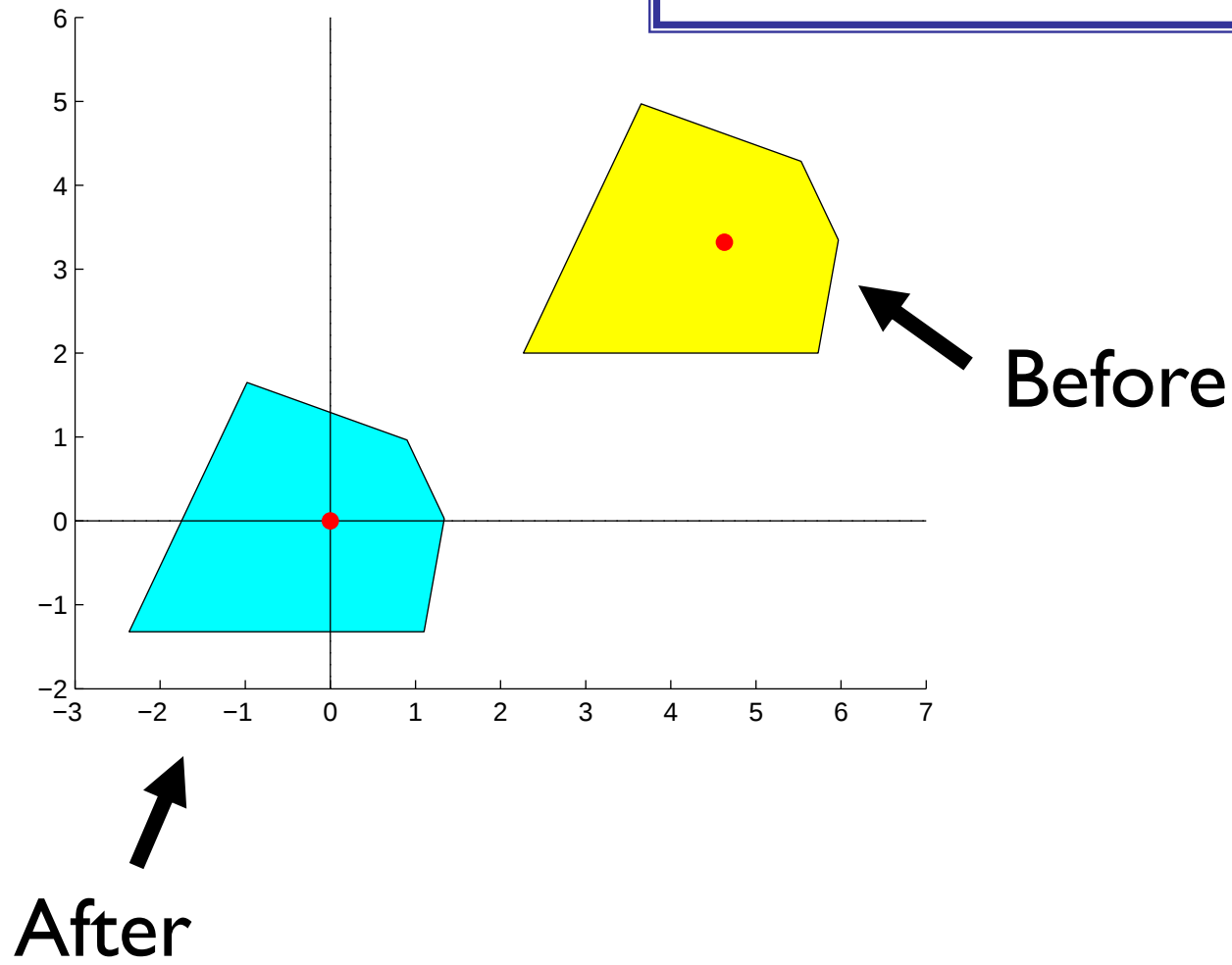


Can store the x-y  
coordinates in  
vectors  $x$  and  $y$

| $x$ | $y$ |
|-----|-----|
|     |     |
|     |     |
|     |     |
|     |     |
|     |     |

# First operation: centralize

Move a polygon so that the centroid of its vertices is at the origin



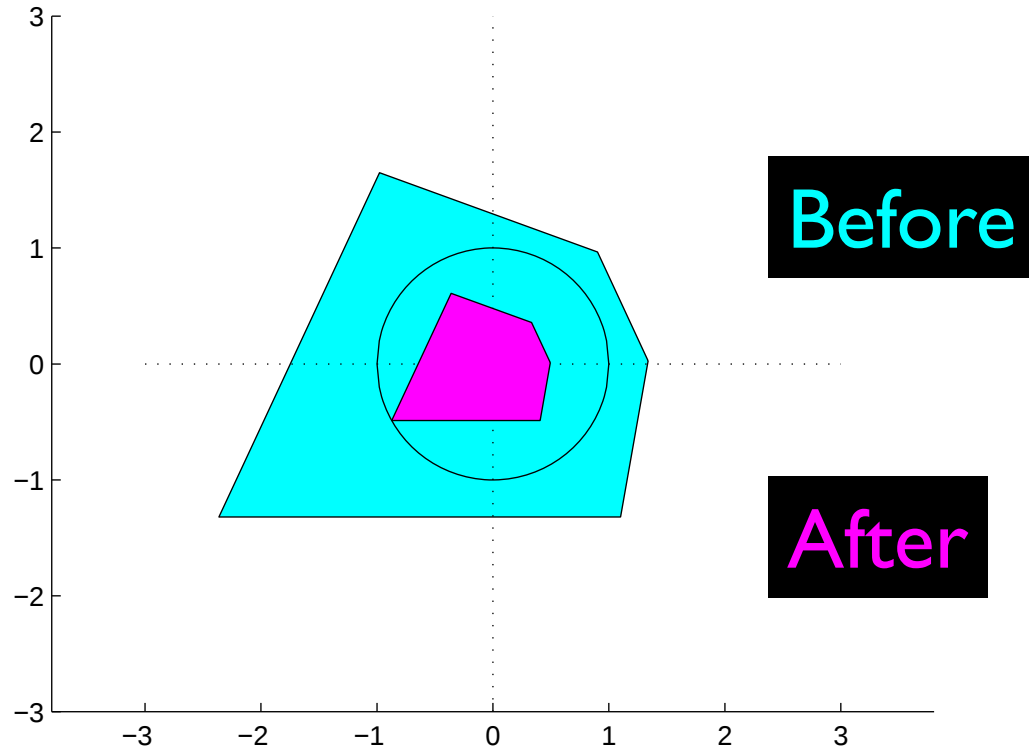
```
function [xNew,yNew] = Centralize(x,y)
% Translate polygon defined by vectors
% x,y such that the centroid is on the
% origin. New polygon defined by vectors
% xNew,yNew.
```

**sum** returns the sum of all values in the vector

```
n = length(x);
xBar = sum(x)/n;    yBar = sum(y)/n;
xNew = zeros(n,1); yNew = zeros(n,1);
for k = 1:n
    xNew(k) = x(k)-xBar;
    yNew(k) = y(k)-yBar;
end
```

## Second operation: normalize

Shrink (enlarge) the polygon so that the vertex furthest from the  $(0,0)$  is on the unit circle



```
function [xNew,yNew] = Normalize(x,y)
% Resize polygon defined by vectors x,y
% such that distance of the vertex
% furthest from origin is 1
```

```
n = length(x);
for k = 1:n
    d(k) = sqrt(x(k)^2 + y(k)^2);
```

```
end
```

```
maxD = max(d);
```

Applied to a vector, **max** returns  
the largest value in the vector

```
xNew = zeros(n,1); yNew = zeros(n,1);
```

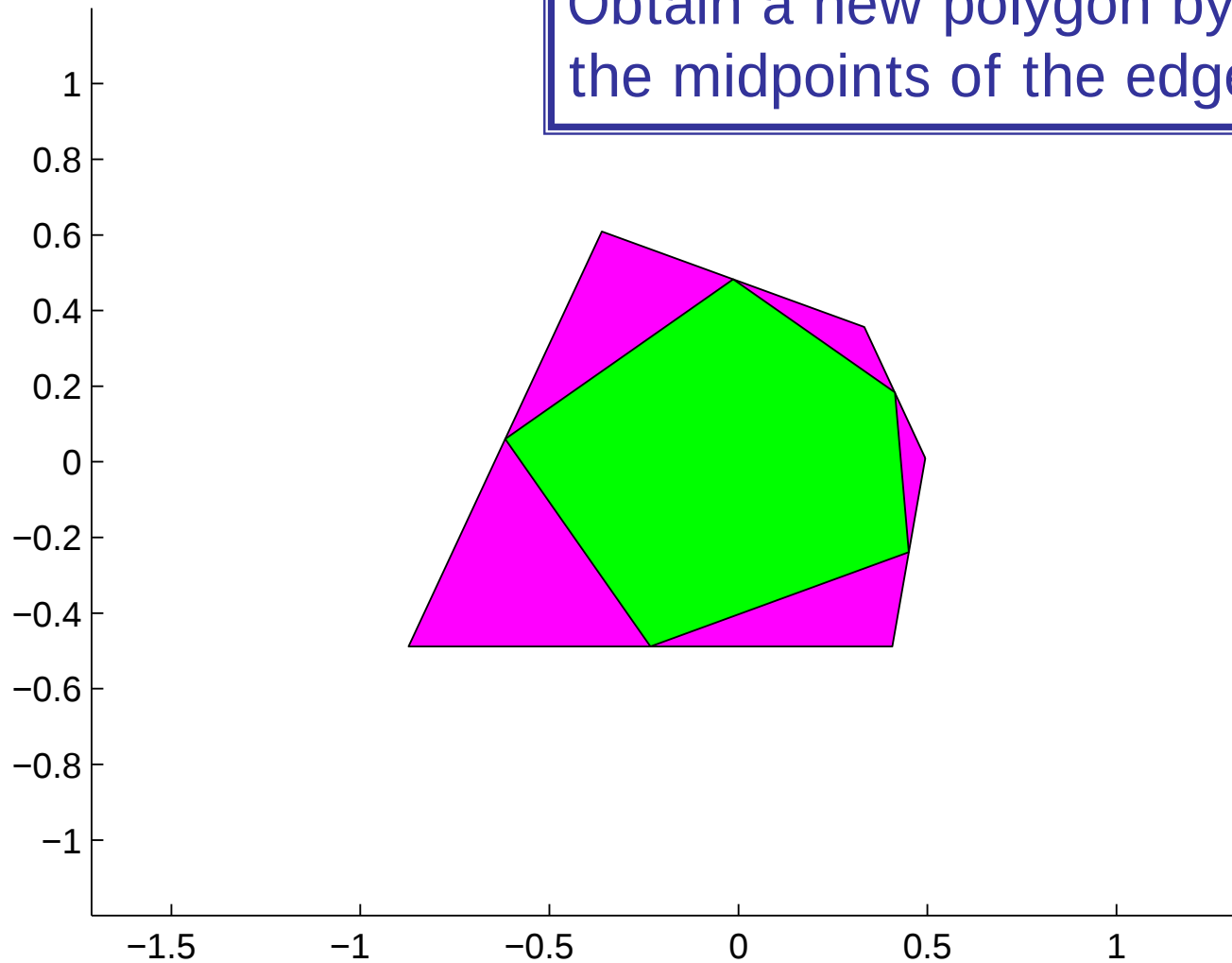
```
for k = 1:n
```

```
    xNew(k)=x(k)/maxD; yNew(k)=y(k)/maxD;
```

```
end
```

## Third operation: smooth

Obtain a new polygon by connecting the midpoints of the edges

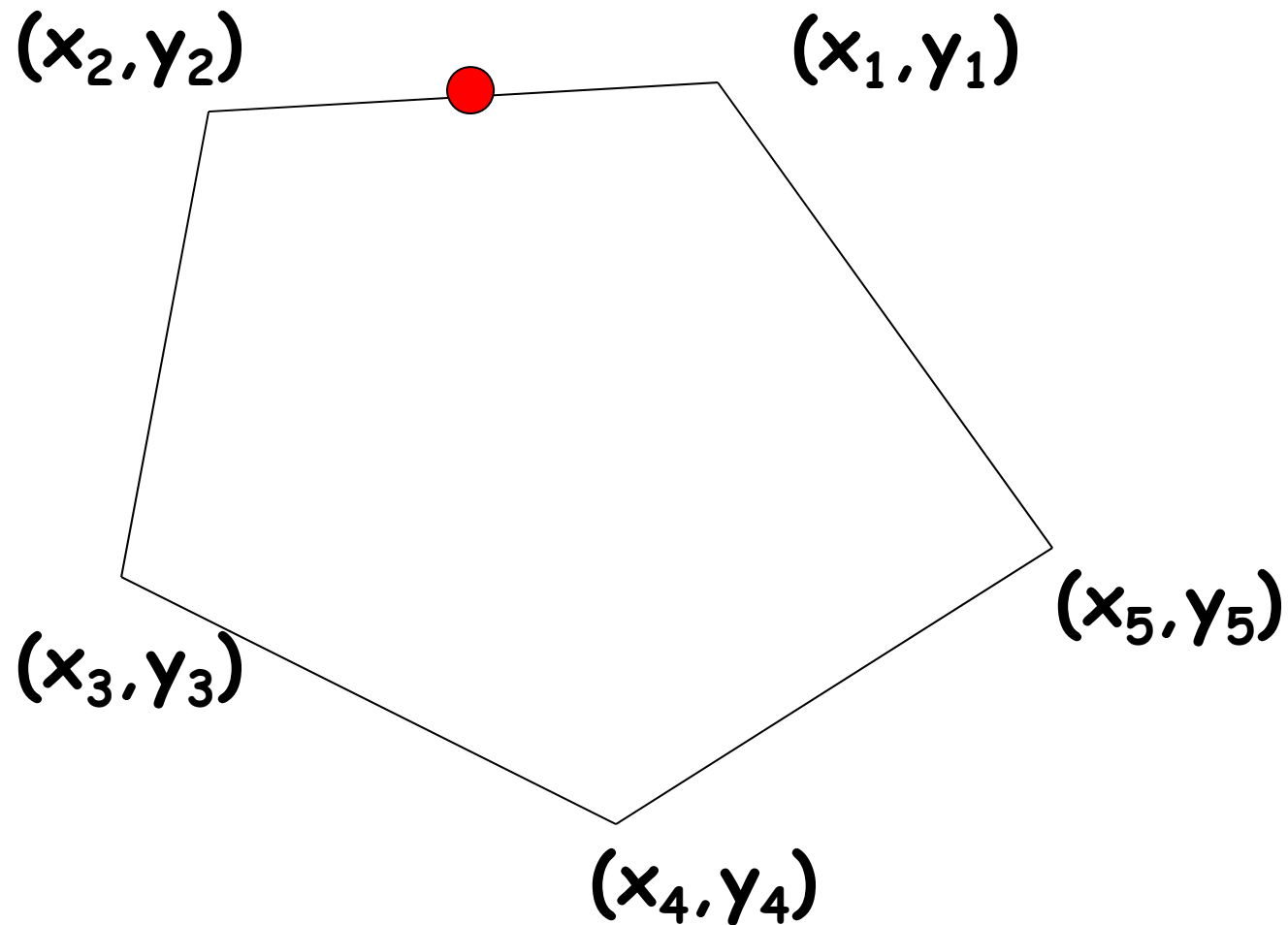


```
function [xNew,yNew] = Smooth(x,y)
% Smooth polygon defined by vectors x,y
% by connecting the midpoints of
% adjacent edges

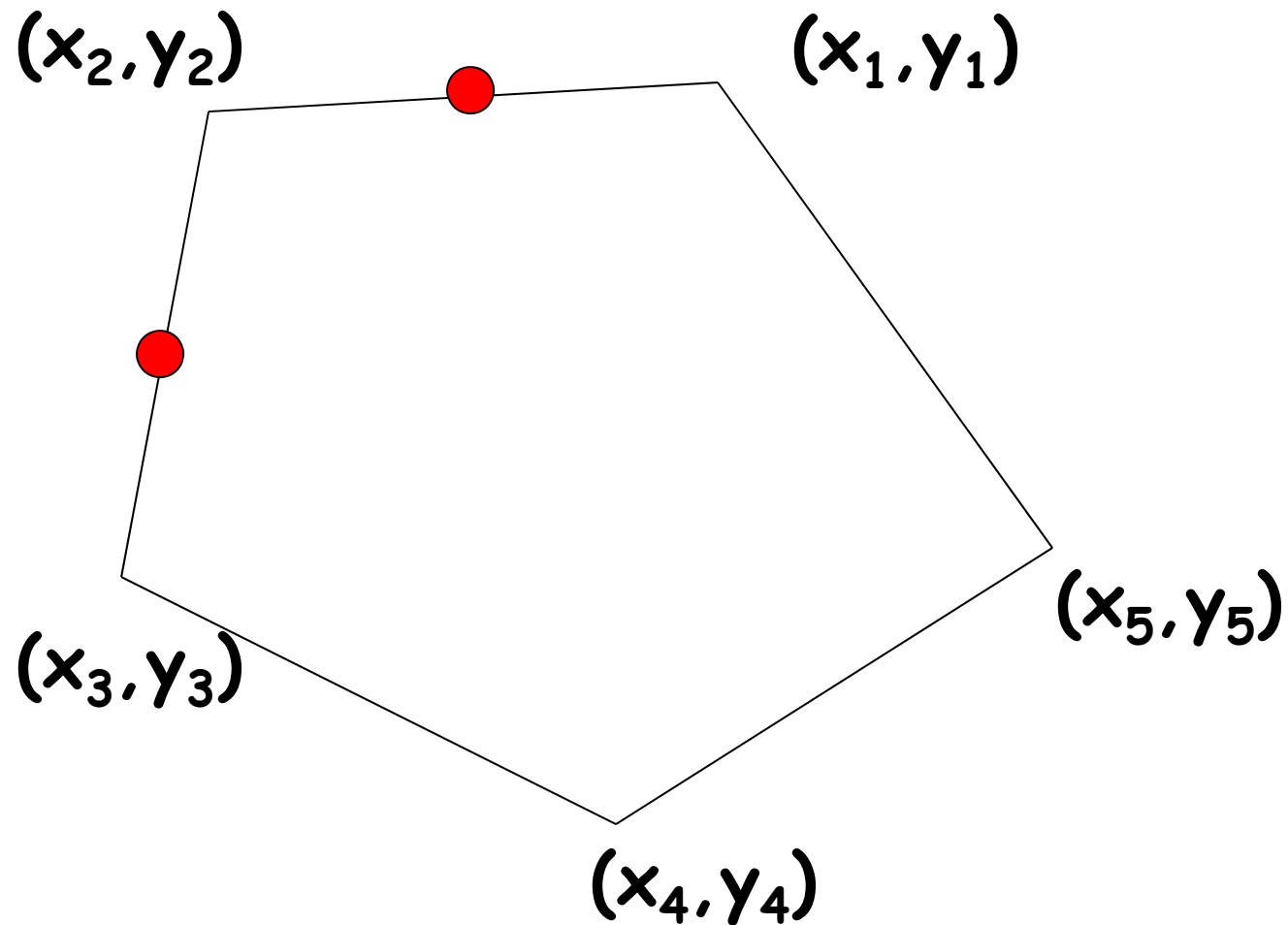
n = length(x);
xNew = zeros(n,1);
yNew = zeros(n,1);
for i=1:n
    %Compute midpt of ith edge. Store in xNew(i), yNew(i)

end
```

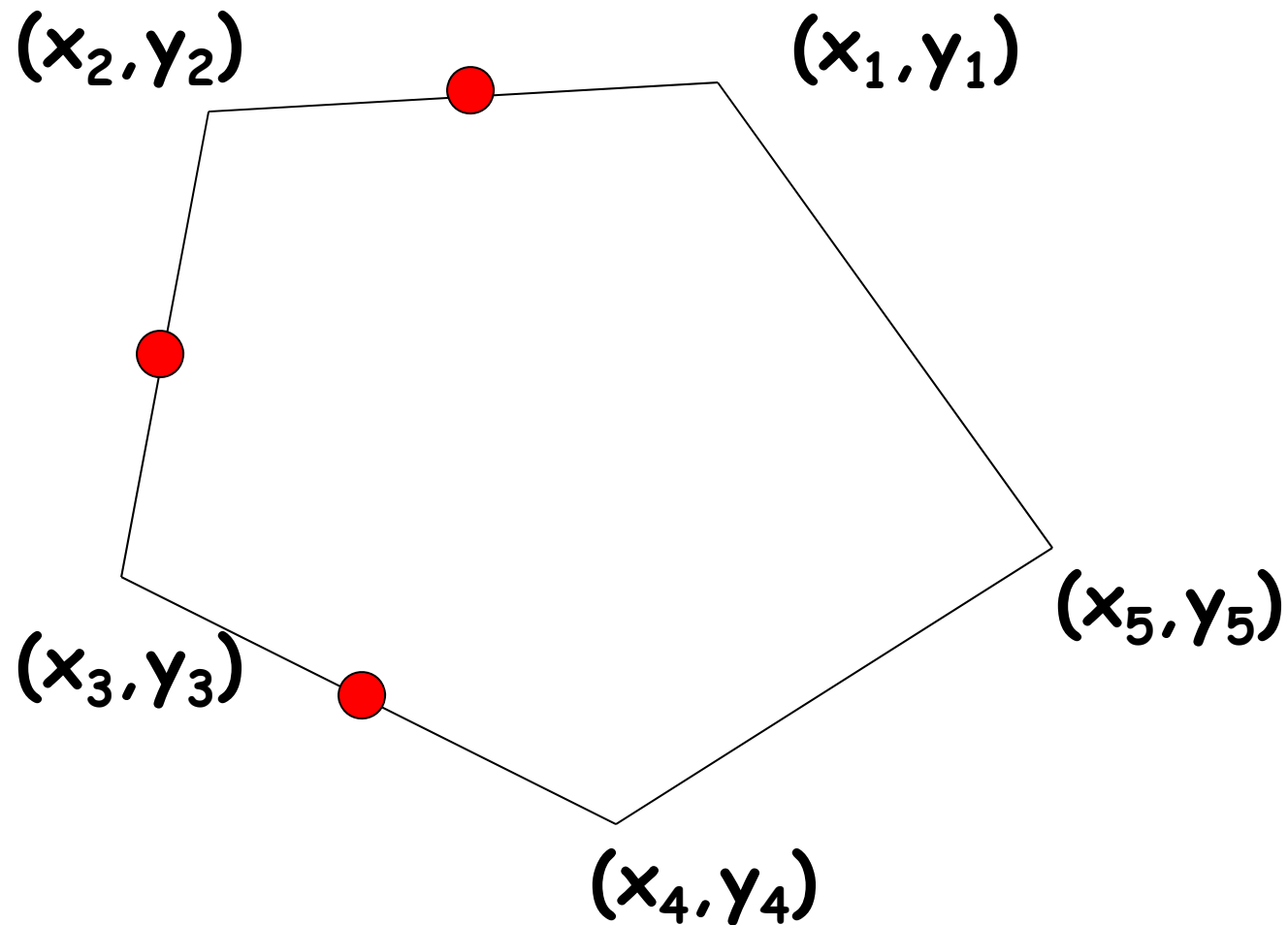
$$\begin{aligned}x_{\text{New}}(\mathbf{1}) &= (x(\mathbf{1}) + x(\mathbf{2})) / 2 \\ y_{\text{New}}(\mathbf{1}) &= (y(\mathbf{1}) + y(\mathbf{2})) / 2\end{aligned}$$



$$\begin{aligned}x_{\text{New}}(2) &= (x(2) + x(3)) / 2 \\ y_{\text{New}}(2) &= (y(2) + y(3)) / 2\end{aligned}$$



$$\begin{aligned}x_{\text{New}}(3) &= (x(3) + x(4)) / 2 \\ y_{\text{New}}(3) &= (y(3) + y(4)) / 2\end{aligned}$$



# Polygon Smoothing

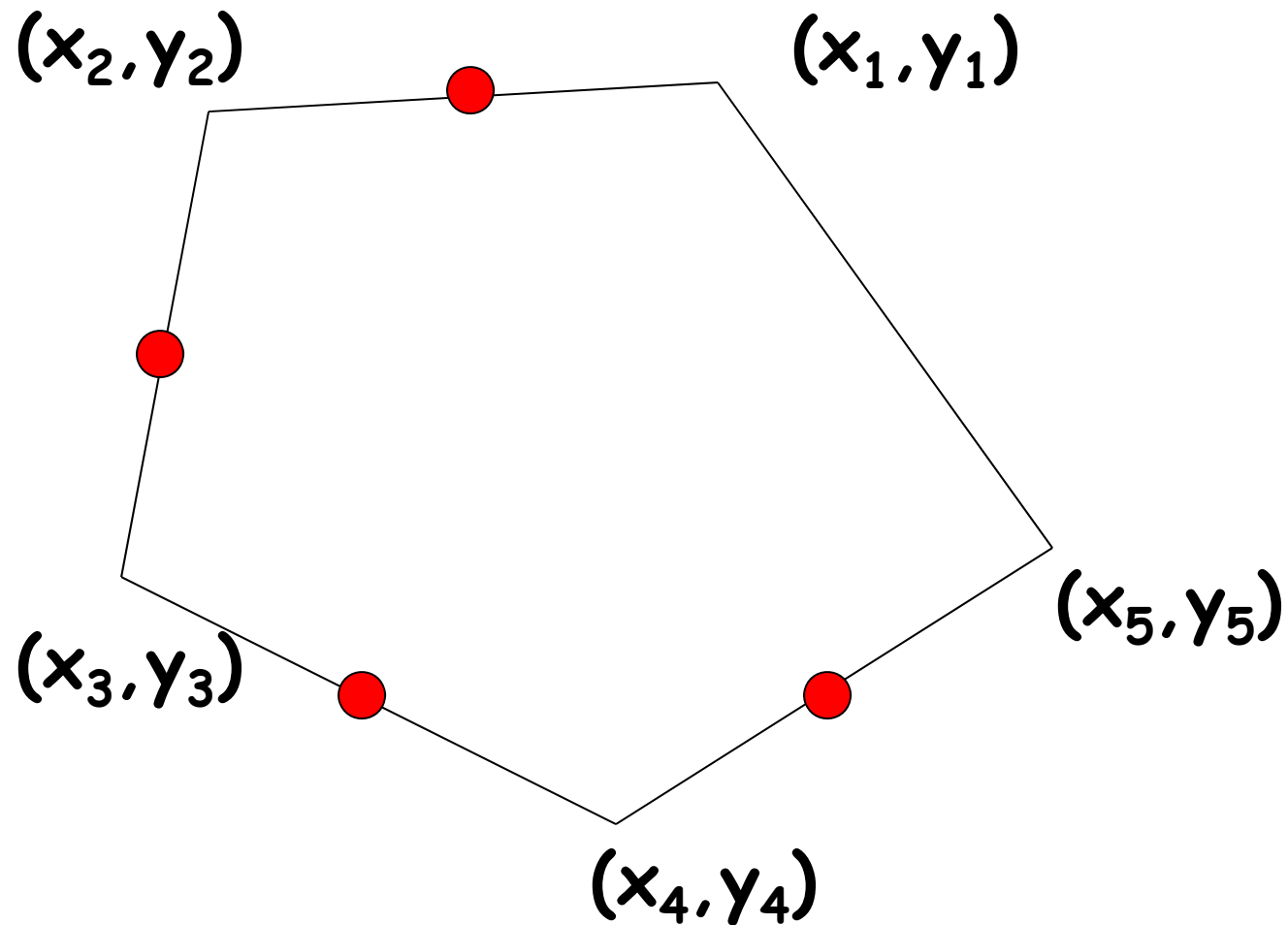
```
% Given n, x, y
for i=1:n
    xNew(i) = (x(i) + x(i+1))/2;
    yNew(i) = (y(i) + y(i+1))/2;
end
```

Does above fragment compute the new n-gon?

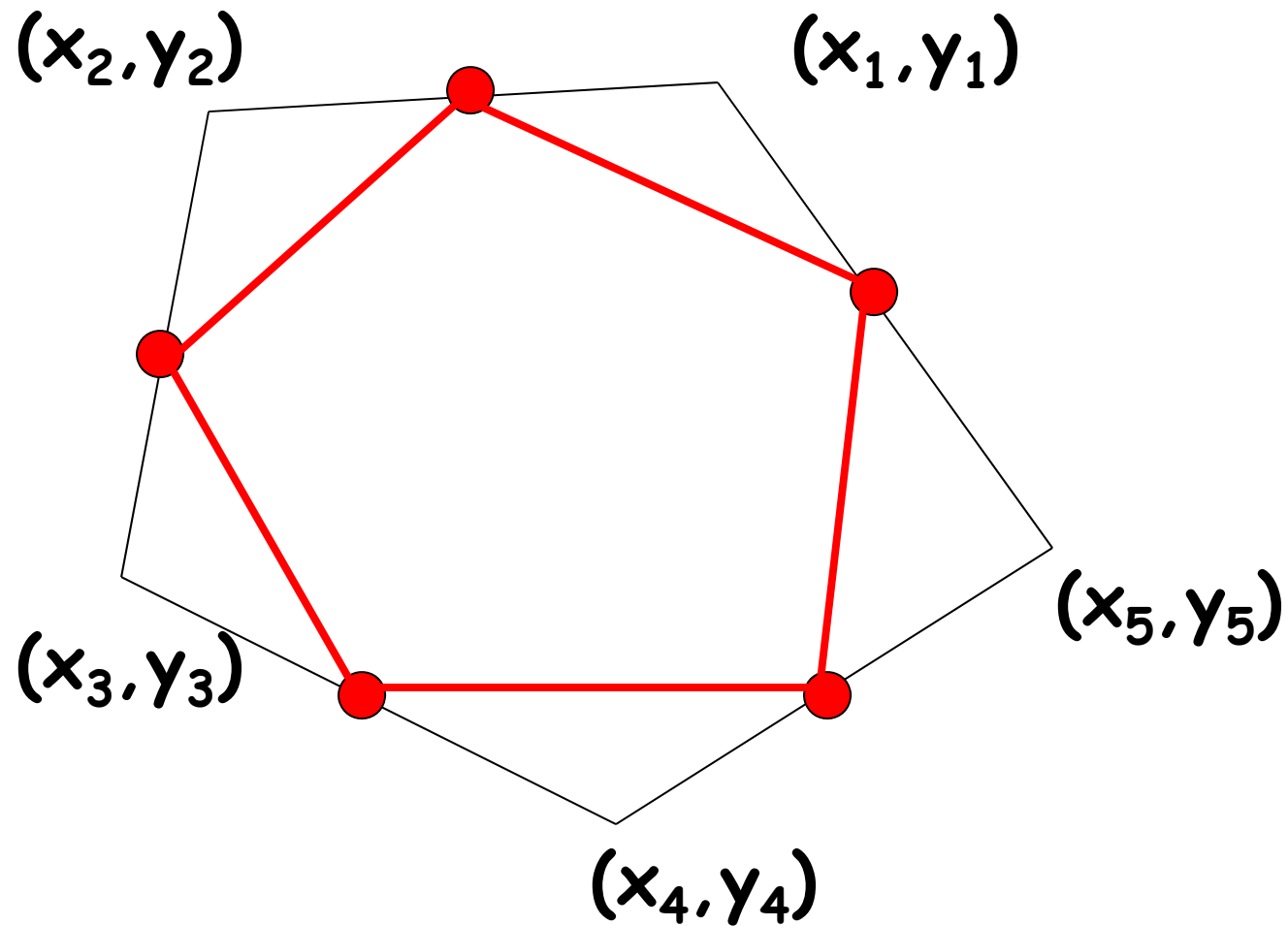
A: Yes

B: No

$$\begin{aligned}x_{\text{New}}(4) &= (x(4) + x(5)) / 2 \\ y_{\text{New}}(4) &= (y(4) + y(5)) / 2\end{aligned}$$



$$\begin{aligned}x_{\text{New}}(5) &= (x(5) + x(1)) / 2 \\ y_{\text{New}}(5) &= (y(5) + y(1)) / 2\end{aligned}$$



## Smooth

```
for i=1:n
    xNew(i) = (x(i) + x(i+1))/2;
    yNew(i) = (y(i) + y(i+1))/2;
end
```

Will result in a subscript  
out of bounds error when i is n.

## Smooth

```
for i=1:n
    if i<n
        xNew(i) = (x(i) + x(i+1))/2;
        yNew(i) = (y(i) + y(i+1))/2;
    else
        xNew(n) = (x(n) + x(1))/2;
        yNew(n) = (y(n) + y(1))/2;
    end
end
```

## Smooth

```
for i=1:n-1
```

```
    xNew(i) = (x(i) + x(i+1))/2;
```

```
    yNew(i) = (y(i) + y(i+1))/2;
```

```
end
```

```
xNew(n) = (x(n) + x(1))/2;
```

```
yNew(n) = (y(n) + y(1))/2;
```

Show a simulation of polygon smoothing

Create a polygon with randomly located vertices.

Repeat:

Centralize

Normalize

Smooth

[See ShowSmooth.m](#)