- Previous Lecture:
  - Finite/inexact arithmetic
  - Plotting continuous functions using vectors and vectorized code
  - Color as a 3-vector

- Today's Lecture:
  - Linear interpolation
  - User-defined functions

- Announcement:
  - Project 2 due tonight at 11pm
  - Prelim 1 Thursday, Oct 8th at 7:30pm. Tell us (email rbhess@cs.cornell.edu) now if you have an exam conflict.

Element-by-element arithmetic operations on arrays…
Also called "vectorized code"

```
x = linspace(-2,3,200);
y = sin(5*x).*exp(-x/2)./(1 + x.^2);
```

x and y are vectors

Contrast with scalar operations that we've used previously…

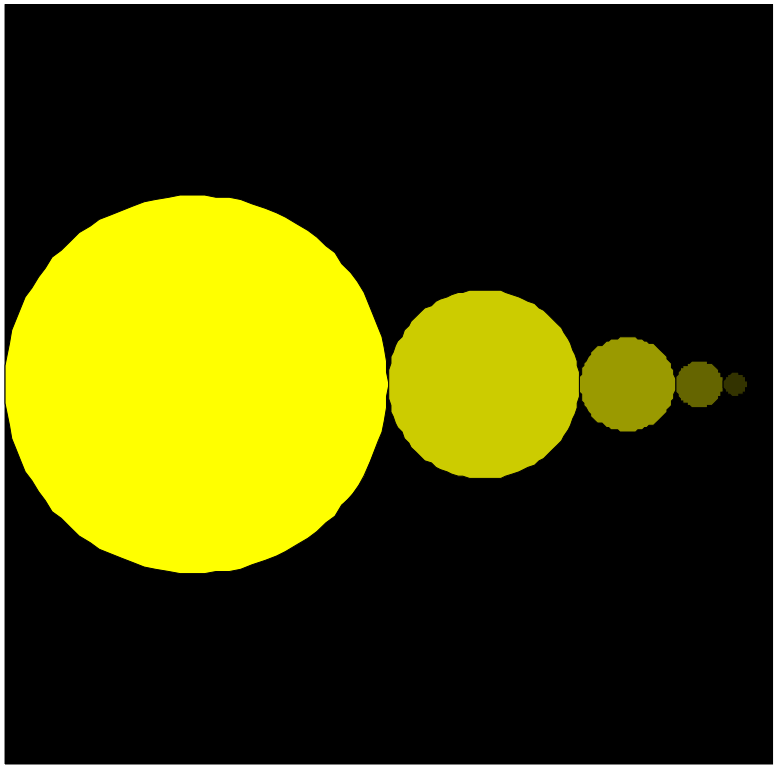
```
a = 2.1;
b = sin(5*a);
```

a and b are scalars

The operators are (mostly) the same; the operands may be scalars or vectors.

When an operand is a vector, you have "vectorized code."

Fading Xeno disks



- First disk is yellow
- Last disk is black (invisible)
- *Interpolate* the color in between

```
% Draw n Xeno disks
d= 1;
x= 0;  % Left tangent point


for k= 1:n


   % Draw kth disk
   DrawDisk(x+d/2, 0, d/2, 'y')
   x= x+d;
   d= d/2;
end
```

```
% Draw n Xeno disks
d= 1;
x= 0;  % Left tangent point


for k= 1:n


   % Draw kth disk
   DrawDisk(x+d/2, 0, d/2, [1 1 0])
   x= x+d;
   d= d/2;
end
```

A vector of length 3

```
% Draw n fading Xeno disks
d= 1;
x= 0;  % Left tangent point
yellow= [1 1 0];
black=  [0 0 0];
for k= 1:n
   % Compute color of kth disk


   % Draw kth disk
   DrawDisk(x+d/2, 0, d/2, _________)
   x= x+d;
   d= d/2;
end
```

Linear interpolation

x	g(x)
:	:
9	110
10	118
11	126
12	134
:	:

Linear interpolation

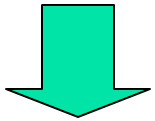
x	g(x)
:	:
9	110
10	118
10.25	?
10.50	?
10.75	?
11	126
12	134
:	:

$$g(10.5) = [\, g(11) + g(10) \,] / 2$$

Linear interpolation

x	g(x)
:	:
9	110
10	118
10.25	?
10.50	?
10.75	?
11	126
12	134
:	:

$$g(10.5) = \tfrac{1}{2}\, g(11) + \tfrac{1}{2}\, g(10)$$



$$g(10.25) = 1/4 \cdot g(11) + 3/4 \cdot g(10)$$
$$g(10.50) = 2/4 \cdot g(11) + 2/4 \cdot g(10)$$
$$g(10.75) = 3/4 \cdot g(11) + 1/4 \cdot g(10)$$

Linear interpolation

x	g(x)
:	:
9	110
10	118
10.25	?
10.50	?
10.75	?
11	126
12	134
:	:

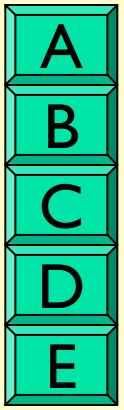
$$g(10.5) = \tfrac{1}{2}\, g(11) + \tfrac{1}{2}\, g(10)$$

$$g(10\quad) = 0/4 \cdot g(11) + 4/4 \cdot g(10)$$
$$g(10.25) = 1/4 \cdot g(11) + 3/4 \cdot g(10)$$
$$g(10.50) = 2/4 \cdot g(11) + 2/4 \cdot g(10)$$
$$g(10.75) = 3/4 \cdot g(11) + 1/4 \cdot g(10)$$
$$g(11\quad) = 4/4 \cdot g(11) + 0/4 \cdot g(10)$$

$$f \cdot g(11) + (1-f) \cdot g(10)$$

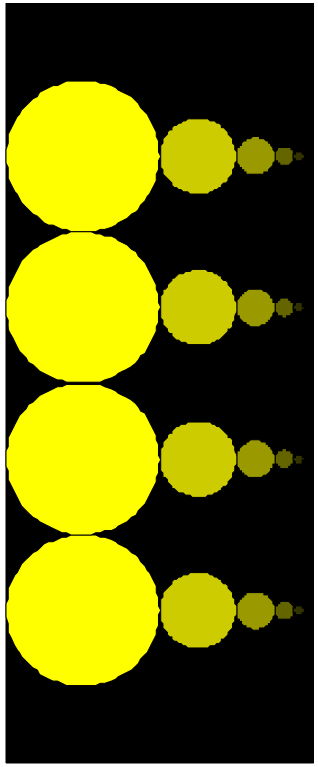
```
% Draw n fading Xeno disks
d= 1;
x= 0;  % Left tangent point
yellow= [1 1 0];
black=  [0 0 0];
for k= 1:n
   % Compute color of kth disk


   % Draw kth disk
   DrawDisk(x+d/2, 0, d/2, ____)
   x= x+d;
   d= d/2;
end
```

```
% Draw n fading Xeno disks
d= 1;
x= 0;  % Left tangent point
yellow= [1 1 0];
black=  [0 0 0];
for k= 1:n
    % Compute color of kth disk
    f= ???
    colr= f*black + (1-f)*yellow;
    % Draw kth disk
    DrawDisk(x+d/2, 0, d/2, colr)
    x= x+d;
    d= d/2;
end
```



k/n
k/(n-1)
(k-1)/n
(k-1)/(n-1)
(k-1)/(n+1)

Rows of Xeno disks



```
for y = __ : __ : __
    Code to draw one
    row of Xeno disks
    at some y-coordinate
end
```

Be careful with "initializations"

```
yellow=[1 1 0];  black=[0 0 0];

d= 1;  x= 0;

for k= 1:n
    % Compute color of kth disk
    f= (k-1)/(n-1);
    colr= f*black + (1-f)*yellow;
    % Draw kth disk
    DrawDisk(x+d/2, 0, d/2, colr)
    x=x+d;  d=d/2;
end
```

```
    yellow=[1 1 0];  black=[0 0 0];
for y= __:__:__
    d= 1;  x= 0; ←necessary for each row

    for k= 1:n
       % Compute color of kth disk
       f= (k-1)/(n-1);
       colr= f*black + (1-f)*yellow;
       % Draw kth disk
       DrawDisk(x+d/2, 0, d/2, colr)
       x=x+d;  d=d/2;
    end
end
```

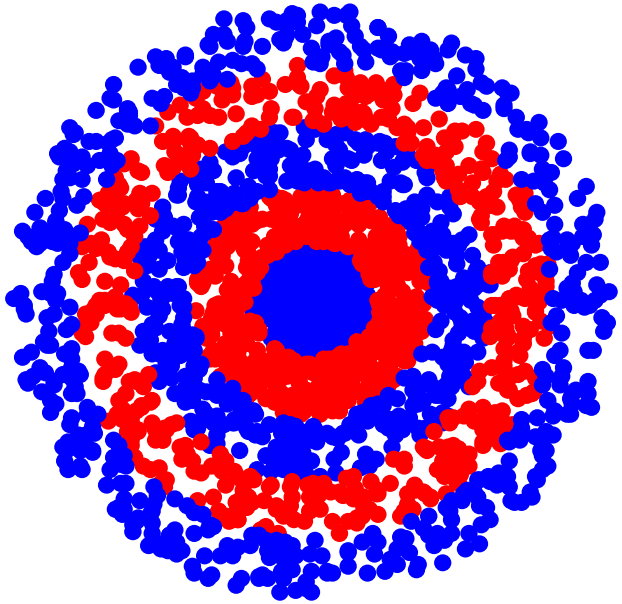
Built-in functions

- We've used many Matlab built-in functions, e.g., **`rand`**, **`abs`**, **`floor`**, **`rem`**
- Example: **`abs(x-.5)`**
- Observations:
  - **`abs`** is set up to be able to work with any valid data
  - **`abs`** doesn't prompt us for input; it expects that we provide data that it'll then work on

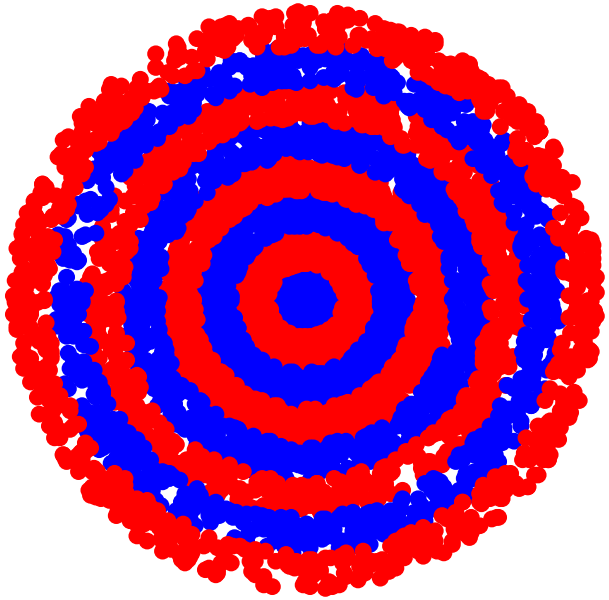
User-defined functions

- We can write our own functions to perform a specific task
  - Example: generate a random floating point number in a specified interval
  - Example: convert polar coordinates to x-y (Cartesian) coordinates

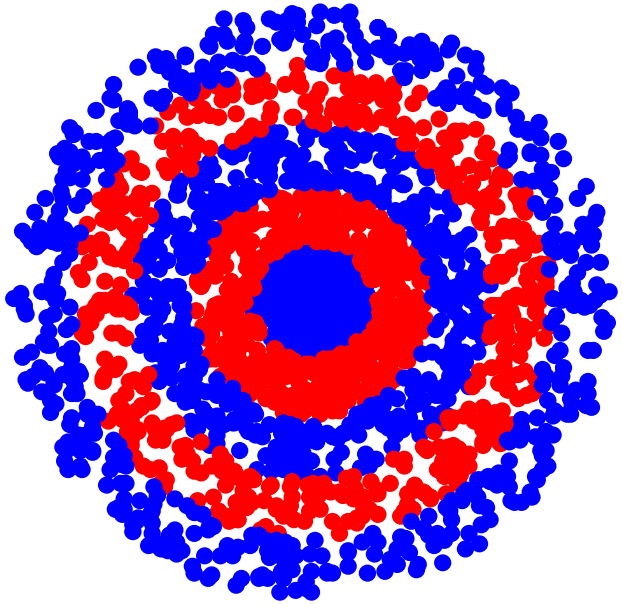
Draw a bulls eye figure with randomly placed dots



- Dots are randomly placed within concentric rings
- User decides how many rings, how many dots



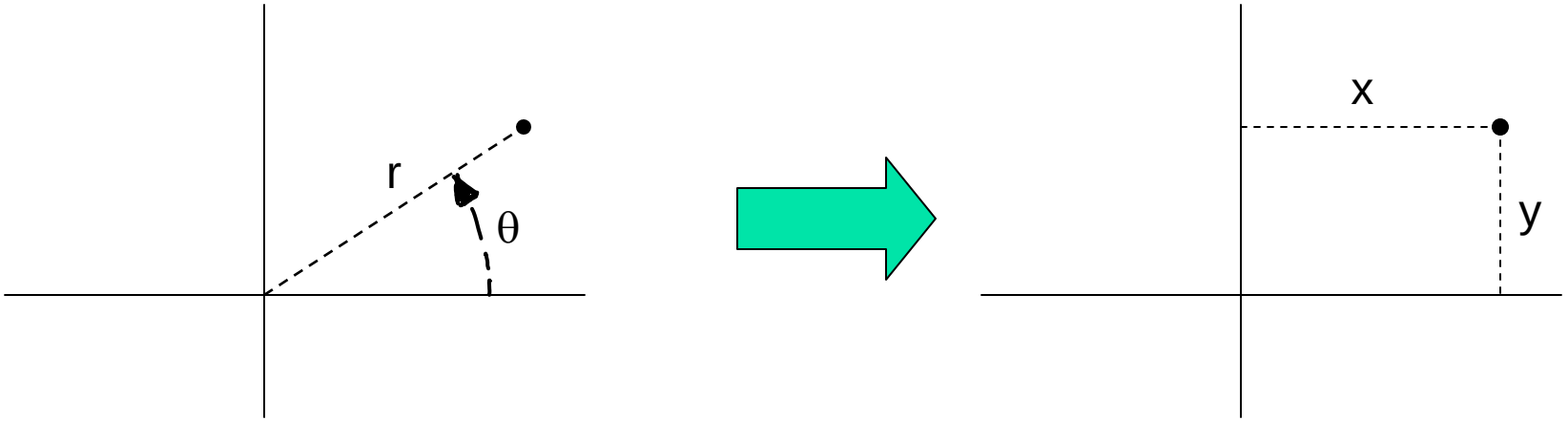
Draw a bulls eye figure with randomly placed dots



- What are the main tasks?
- Accommodate variable number of rings—loop

- For each ring
  - Need many dots
  - For each dot
    - Generate random position
    - Choose color
    - Draw it

Convert from polar to Cartesian coordinates



Polar coordinates

Cartesian coordinates

```
c= input('How many concentric rings? ');
d= input('How many dots? ');

% Put dots btwn circles with radii rRing and (rRing-1)
for rRing= 1:c
  % Draw d dots
  for count= 1:d

    % Generate random dot location (polar coord.)
    theta=_______
    r=_______

    % Convert from polar to Cartesian
    x=_______
    y=_______

    % Use plot to draw dot
  end
end
```

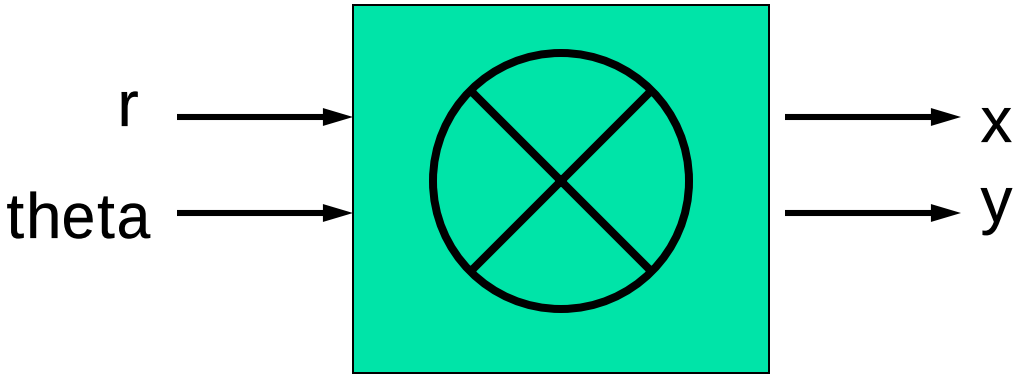
A common task! Create a function **polar2xy** to do this. **polar2xy** likely useful in other problems as well.

```
function [x, y] = polar2xy(r,theta)
% Convert polar coordinates (r,theta) to
% Cartesian coordinates (x,y).
% theta is in degrees.

rads= theta*pi/180;  % radian
x= r*cos(rads);
y= r*sin(rads);
```

A function file **polar2xy.m**

```
function [x, y] = polar2xy(r,theta)
% Convert polar coordinates (r,theta) to
% Cartesian coordinates (x,y).
% theta is in degrees.

rads= theta*pi/180;  % radian
x= r*cos(rads);
y= r*sin(rads);
```

Think of **`polar2xy`** as a factory



```
function [x, y] = polar2xy(r,theta)
% Convert polar coordinates (r,theta) to
% Cartesian coordinates (x,y).
% theta is in degrees.

rads= theta*pi/180;  % radian
x= r*cos(rads);
y= r*sin(rads);
```

A function file **polar2xy.m**

```
r= input('Enter radius: ');
theta= input('Enter angle in degrees: ');

rads= theta*pi/180;  % radian
x= r*cos(rads);
y= r*sin(rads);
```

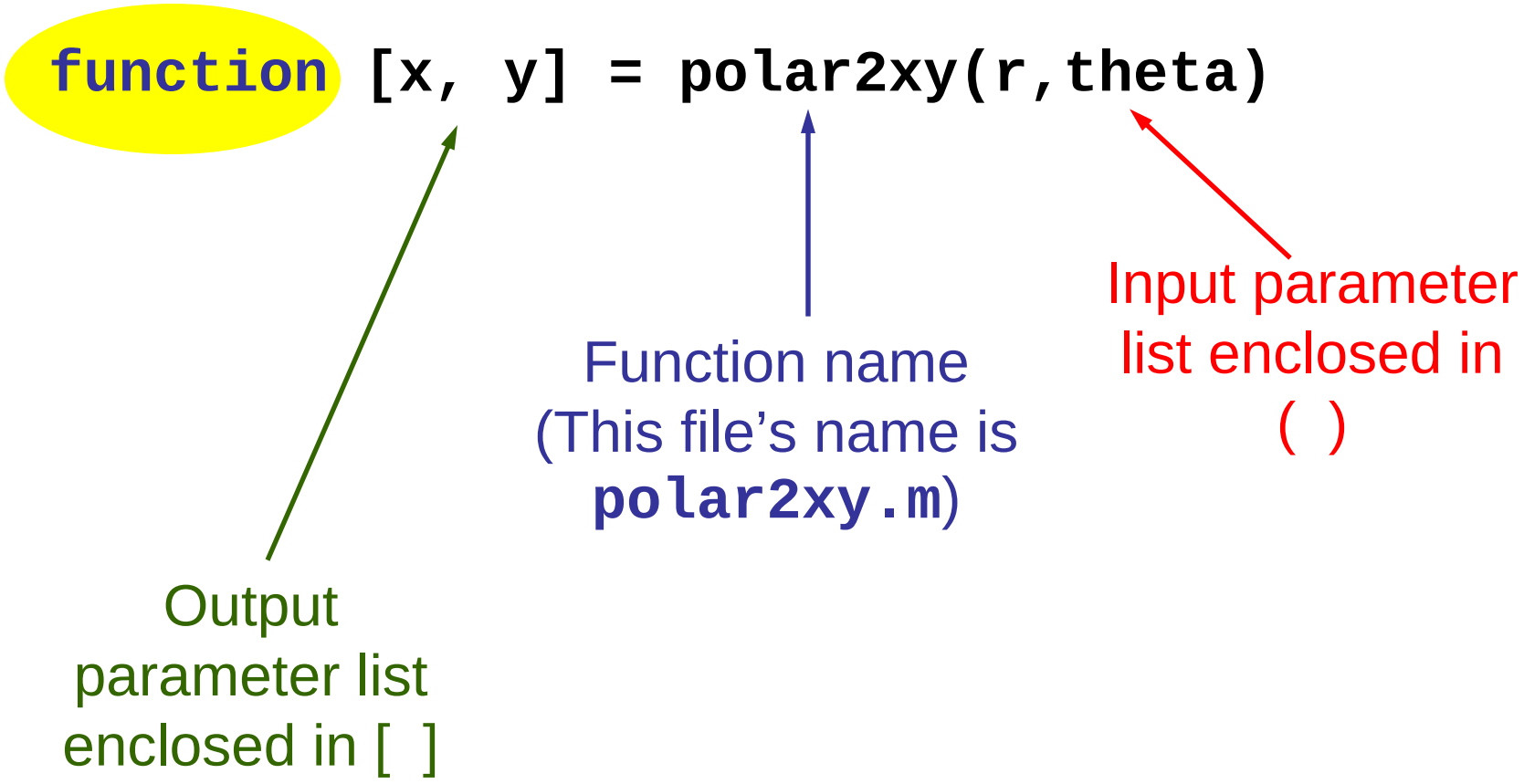
(Part of) a script file

```
function [x, y] = polar2xy(r,theta)
% Convert polar coordinates (r,theta) to
% Cartesian coordinates (x,y).
% theta is in degrees.

rads= theta*pi/180;  % radian
x= r*cos(rads);
y= r*sin(rads);
```

A function file **polar2xy.m**

```
r= input('Enter radius: ');
theta= input('Enter angle in degrees: ');

rads= theta*pi/180;  % radian
x= r*cos(rads);
y= r*sin(rads);
```

(Part of) a script file


function [x, y] = polar2xy(r,theta)
Output parameter list enclosed in [ ]
Function name (This file's name is polar2xy.m)
Input parameter list enclosed in ( )

Function header is the "contract" for how the function will be used (called)

You have this function:

```
function  [x, y] = polar2xy(r, theta)
% Convert polar coordinates (r, theta) to
% Cartesian coordinates (x,y).  Theta in degrees.
…
```

Code to call the above function:

```
% Convert polar (r1,t1) to Cartesian (x1,y1)
r1= 1;   t1= 30;
[x1, y1]= polar2xy(r1, t1);
plot(x1, y1, 'b*')
…
```

Function header is the "contract" for how the function will be used (called)

You have this function:

```
function  [x, y] = polar2xy(r, theta)
% Convert polar coordinates (r, theta) to
% Cartesian coordinates (x,y).  Theta in degrees.
…
```

Code to call the above function:

```
% Convert polar (r1,t1) to Cartesian (x1,y1)
r1= 1;   t1= 30;
[x1, y1]= polar2xy(r1, t1);
plot(x1, y1, 'b*')
…
```

Function header is the "contract" for how the function will be used (called)

You have this function:

```
function  [x, y] = polar2xy(r, theta)
% Convert polar coordinates (r,theta) to
% Cartesian coordinates (x,y).  Theta in degrees.
...
```

Code to call the above function:

```
% Convert polar (r1,t1) to Cartesian (x1,y1)
r1= 1;  t1= 30;
[x1, y1]= polar2xy(r1, t1);
plot(x1, y1, 'b*')
...
```

General form of a user-defined function

```
function [out1, out2, …]= functionName (in1, in2, …)
% 1-line comment to describe the function
% Additional description of function

    Executable code that at some point assigns
    values to output parameters out1, out2, …
```

- *in1*, *in2*, … are defined when the function begins execution. Variables *in1*, *in2*, … are called function *parameters* and they hold the function *arguments* used when the function is invoked (called).
- *out1*, *out2*, … are not defined until the executable code in the function assigns values to them.

Comments in functions

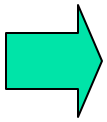
- Block of comments after the function header is printed whenever a user types

  `help <functionName>`

  at the Command Window
- 1st line of this comment block is searched whenever a user types

  `lookfor <someWord>`

  at the Command Window



- Every function should have a comment block after the function header that says what the function does concisely

dotsInCircles.m

(functions with multiple input parameters)

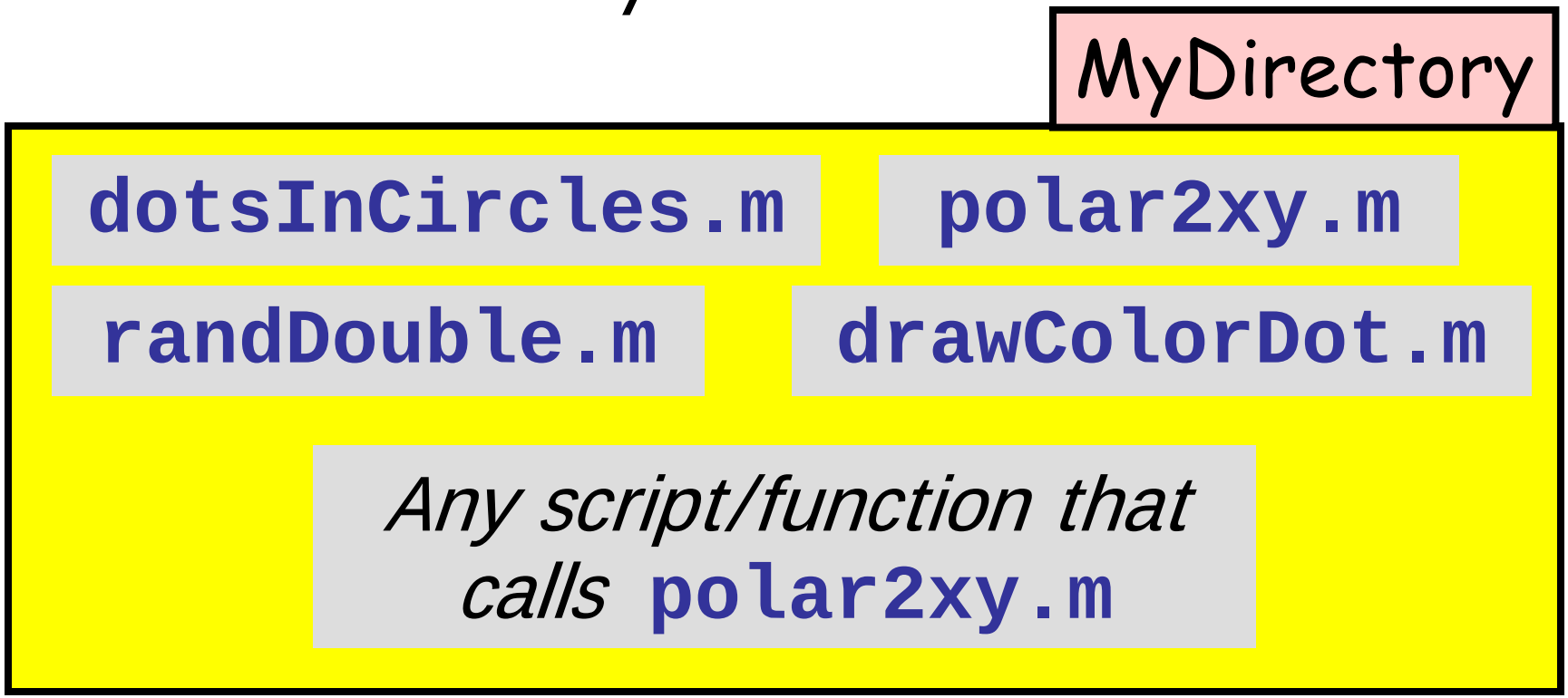
(functions with a single output parameter)

(functions with multiple output parameters)

(functions with no output parameter)

Accessing your functions

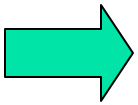
For now*, put your related functions and scripts in the same directory.



*The **path** function gives greater flexibility

Why write user-defined function?

- Easy code re-use—great for "common" tasks
- A function can be tested independently easily
- Keep a driver program clean by keeping detail code in functions—separate, non-interacting files



Facilitate top-down design