

- Previous Lecture:

- Nested loops
- Developing algorithms and code

- Today's Lecture:

- Review nested loops
- User-defined functions

- Announcements:

- Project 2 due today at 11pm
- This weekend is a great time to review, get caught up

Rational approximation of π

- $\pi = 3.141592653589793\dots$
- Can be closely approximated by fractions, e.g., $\pi \approx 22/7$
- Rational number: a quotient of two integers
- Approximate π as p/q where p and q are positive integers $\leq M$
- Start with a straight forward solution:
 - Get M from user
 - Calculate quotient p/q for all combinations of p and q
 - Pick best quotient \rightarrow smallest error

```
% Rational approximation of pi
```

```
M = input('Enter M: ');
```

```
% Check all possible denominators
```

```
% Rational approximation of pi
```

```
M = input('Enter M: ');
```

```
% Check all possible denominators
```

```
for q = 1:M
```

```
end
```

```
% Rational approximation of pi
```

```
M = input('Enter M: ');
```

```
% Check all possible denominators
```

```
for q = 1:M
```

For current q find best numerator p...
Check all possible numerators

```
end
```

```
% Rational approximation of pi

M = input('Enter M: ');

% Check all possible denominators
for q = 1:M
    % At this q, check all possible numerators
    for p = 1:M

        end
    end
end
```

```
% Rational approximation of pi

M = input('Enter M: ');
% Best q, p, and error so far
qBest=1;  pBest=1;
err_pq = abs(pBest/qBest - pi);

% Check all possible denominators
for q = 1:M
    % At this q, check all possible numerators
    for p = 1:M

        end

    end

end

myPi = pBest/qBest;
```

```

% Rational approximation of pi

M = input('Enter M: ');
% Best q, p, and error so far
qBest=1;  pBest=1;
err_pq = abs(pBest/qBest - pi);

% Check all possible denominators
for q = 1:M
    % At this q, check all possible numerators
    for p = 1:M
        if abs(p/q - pi) < err_pq % best p/q found
            err_pq = abs(p/q - pi);
            pBest= p;
            qBest= q;
        end
    end
end

myPi = pBest/qBest;

```

```
% Complicated version in the book
```

```
M = input('Enter M: ');
```

```
% Best q, p, and error so far
```

```
qBest=1;  pBest=1;
```

```
err_pq = abs(pBest/qBest - pi);
```

```
% Check all possible denominators
```

```
for q = 1:M
```

```
    % At this q, check all possible numerators
```

```
    p0=1;  e0=abs(p0/q - pi); % best p & error for this q
```

```
    for p = 1:M
```

```
        if abs(p/q - pi) < e0 % new best numerator found
```

```
            p0=p;  e0 = abs(p/q - pi);
```

```
        end
```

```
    end
```

```
    % Is best quotient for this q is best over all?
```

```
    if e0 < err_pq
```

```
        pBest=p0;  qBest=q;  err_pq=e0;
```

```
    end
```

```
end
```

```
myPi = pBest/qBest;
```

```
% Rational approximation of pi
```

```
M = input('Enter M: ');
```

```
% Best q, p, and error so far
```

```
qBest=1; pBest=1;
```

```
err_pq = abs(pBest/qBest - pi);
```

```
% Check all possible denominators
```

```
for q = 1:M
```

```
    % At this q, check all possible numerators
```

```
    for p = 1:M
```

```
        if abs(p/q - pi) < err_pq % best p/q found
```

```
            err_pq = abs(p/q - pi);
```

```
            pBest= p;
```

```
            qBest= q;
```

```
        end
```

```
    end
```

```
end
```

```
myPi = pBest/qBest;
```

Algorithm: Finding the best in a set

Init bestSoFar

Loop over set

 if current is better than bestSoFar

 bestSoFar \leftarrow current

 end

end

Analyze the program for efficiency

- See Eg3_1 and FasterEg3_1 in the book

```
for a = 1:n
    disp('alpha')
    for b = 1:m
        disp('beta')
    end
end
```

How many times are “alpha”
and “beta” displayed?

A: n, m

B: m, n

C: $n, n+m$

D: $n, n*m$

E: $m*n, m$

Built-in functions

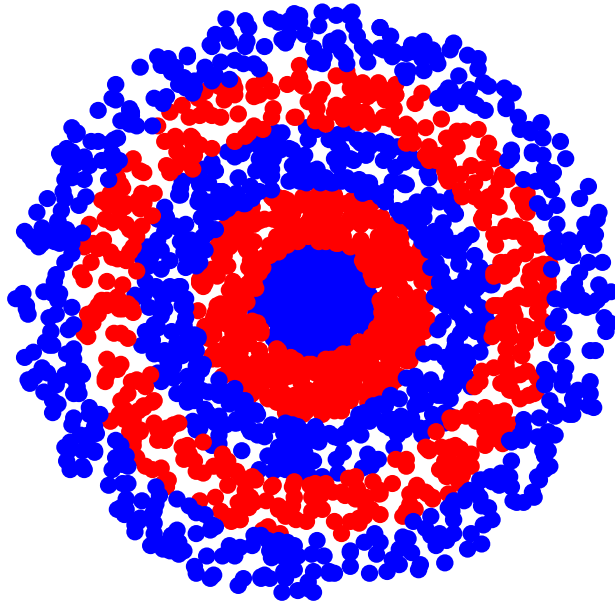
- We've used many Matlab built-in functions, e.g., **rand**, **abs**, **floor**, **rem**
- Example: **abs(x - .5)**
- Observations:
 - **abs** is set up to be able to work with any valid data
 - **abs** doesn't prompt us for input; it expects that we provide data that it'll then work on
 - **abs** returns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

```
yDistance= abs(y2-y1);  
  
while abs(myPi-pi) > .0001  
    ...
```

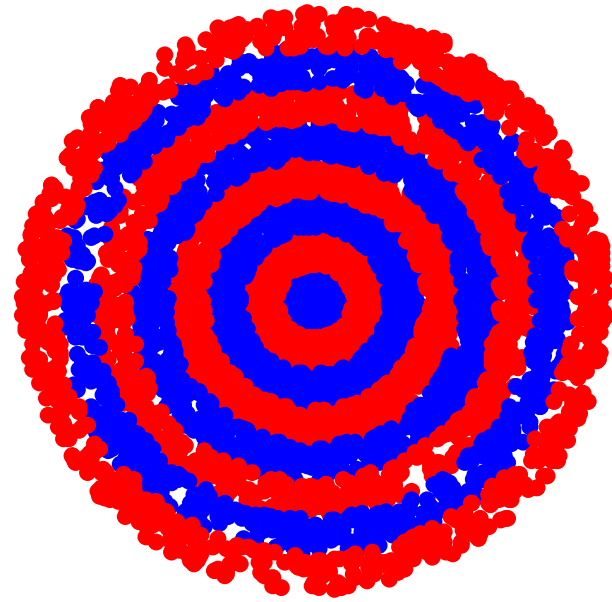
User-defined functions

- We can write our own functions to perform a specific task
 - **Example:** draw a disk with specified radius, color, and center coordinates
 - **Example:** generate a random floating point number in a specified interval
 - **Example:** convert polar coordinates to x-y (Cartesian) coordinates

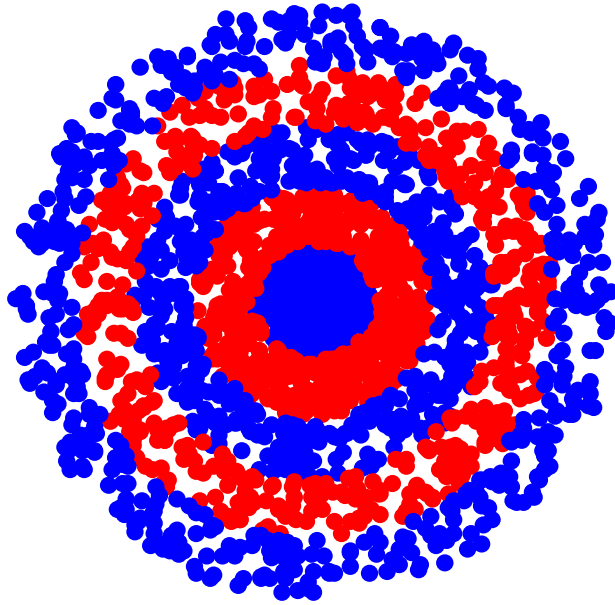
Draw a bulls eye figure with randomly placed dots



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

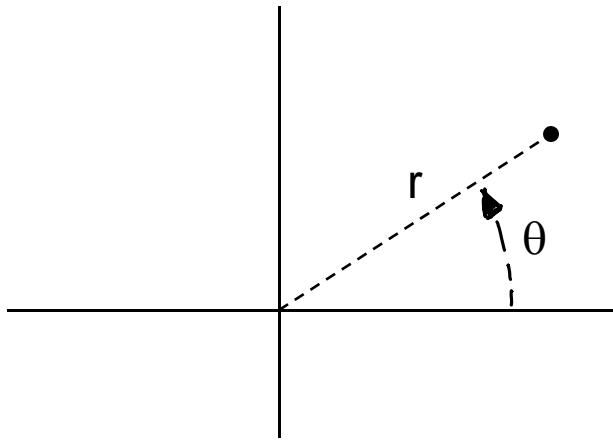


Draw a bulls eye figure with randomly placed dots

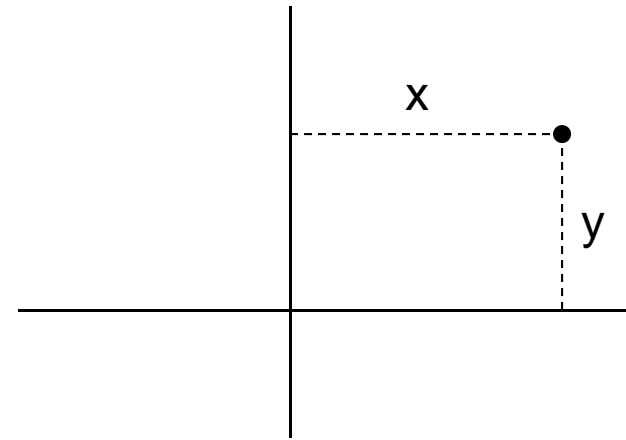
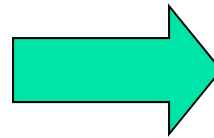


- What are the main tasks?
- Accommodate variable number of rings—loop
- For each ring
 - Need many dots
 - For each dot
 - Generate random position
 - Choose color
 - Draw it

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
end

```

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

```

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

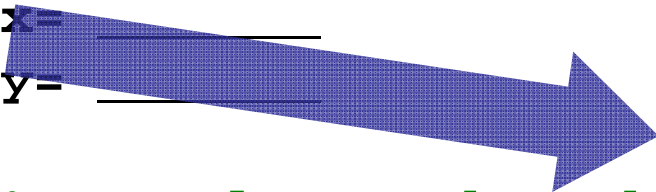
% Put dots btwn circles with radii rRing and (rRing-1)
for rRing= 1:c
    % Draw d dots
    for count= 1:d

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

        % Convert from polar to Cartesian
        x= _____
        y= _____
        [x,y] = polar2xy(r,theta);

        % Use plot to draw dot
    end
end
end

```



```
% Generate random dot location (polar)
```

```
theta= _____
```

```
r= _____
```

```
% Convert from polar to Cartesian
```

```
rads= theta*pi/180; % radian
```

```
x= r*cos(rads);
```

```
y= r*sin(rads);
```

Part of a script

```
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);
```

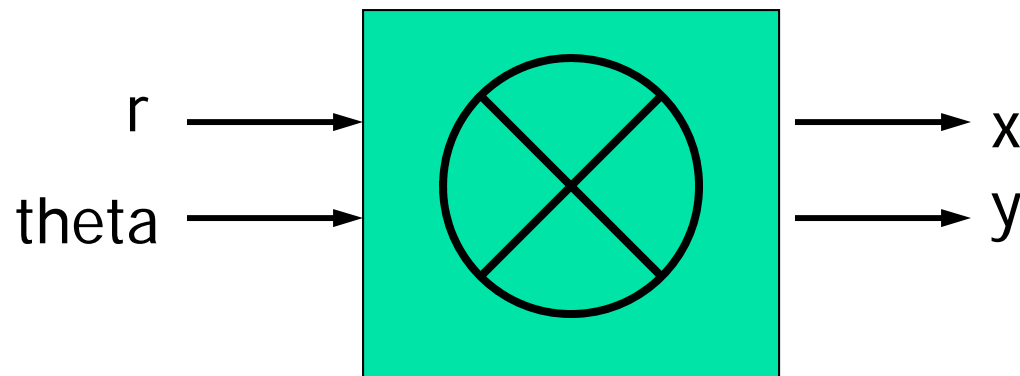
A function file
polar2xy.m

```
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);
```

A function file
`polar2xy.m`

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

```

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
end

```

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

```

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= _____
        [x,y] = polar2xy(r,theta);

        % Use plot to draw dot
    end
end
end

```

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*')  
...
```

dotsInRings.m

(functions with multiple input parameters)

(functions with a single output parameter)

(functions with multiple output parameters)

(functions with no output parameter)

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.