

- 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
for q = 1:M

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

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;
```

Algorithm: Finding the best in a set

```
Init bestSoFar
Loop over set
  if current is better than bestSoFar
    bestSoFar ← current
  end
end
```

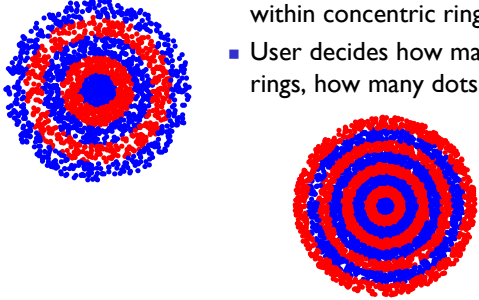
- ### Built-in functions
- We've used many Matlab built-in functions, e.g., **rand**, **abs**, **floor**, **rem**
 - Example: **abs(x-.5)**
 - Observations:
 - **abs** is set up to be able to work with any valid data
 - **abs** *doesn't prompt us for input; it expects that we provide data* that it'll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

Lecture 8 16

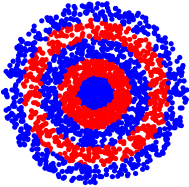
### Draw a bulls eye figure with randomly placed dots



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

Lecture 8 18

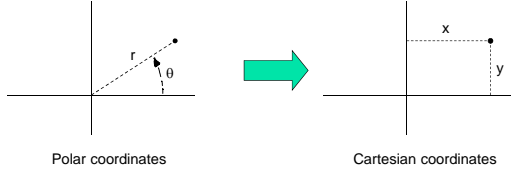
### Draw a bulls eye figure with randomly placed dots



- 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

Lecture 8 19

### Convert from polar to Cartesian coordinates



Lecture 8 20

```

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 will be useful in other problems as well.

Lecture 8 21

```

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

```

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

Lecture 8 22

```
% 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

Lecture 8 25

```
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

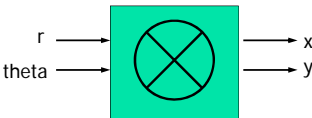
Lecture 8 26

```
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



Lecture 8 27

```
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);
```

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

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

A function file  
polar2xy.m

(Part of) a  
script file

Lecture 8 28

```
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 ( )

Lecture 8 32

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*')
```

Lecture 8 34

dotsInRings.m

(functions with multiple input parameters)  
 (functions with a single output parameter)  
 (functions with multiple output parameters)  
 (functions with no output parameter)

Lecture 8 36

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.

Lecture 8 37

Returning a value ≠ printing a value

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*')
...
```

Lecture 8 38

Comments in functions

- Block of **comments after the function header** is printed whenever a user types `help <functionName>` at the Command Window
- **1<sup>st</sup> 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**

Lecture 8 40

Given this function:

```
function m = convertLength(ft,in)
% Convert length from feet (ft) and inches (in)
% to meters (m).
...
```

How many proper calls to `convertLength` are shown below?

```
% Given f and n
d= convertLength(f,n);
d= convertLength(f*12+n);
d= convertLength(f+n/12);
x= min(convertLength(f,n), 1);
y= convertLength(pi*(f+n/12)^2);
```

A: 1   
  B: 2   
  C: 3   
  D: 4   
  E: 5 or 0

Lecture 8 39

Accessing your functions

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

MyDirectory

dotsInCircles.m
polar2xy.m

randDouble.m
drawColorDot.m

Any script/function that calls `polar2xy.m`

\*The `path` function gives greater flexibility

Lecture 8 41