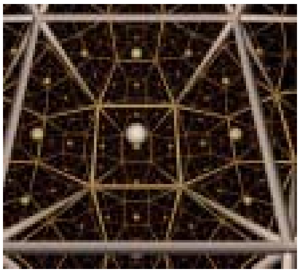


Programming with Vectors & Strings

Lecture 13 (Mar 4)
CS100M – Spring 2008

Announcements

- Section this week is in the classroom (not the lab)
- Prelim 2 is coming soon!
  - Date: Thursday, March 13
  - Time: 7:30-9:00 pm
  - If you have a conflict, tell us (email Kelly Patwell) today
    - We accommodate only university-accepted conflicts
    - Leaving early for spring break doesn't count

Characters ↔ ASCII Code

```
str = 'CS100M';          % Vector (1D array) of characters

code = double(str);      % Converts string into vector of numbers

s = char(code);          % Converts vector of numbers into a string
```

Character Arithmetic

- You can do “math” with characters

```
'd' – 'a'          % Produces 3
'9' – '8'          % Produces 1
'a' < 'd'          % Produces 1 (= true)
'd' < 'b'          % Produces 0 (= false)
'Z' < 'b'          % Produces 1 (= true)
                   % Because 90, the ASCII code for 'Z',
                   % is less than 98, the ASCII code for 'b'

'a' + 2            % Produces 99
char('a'+2)        % Produces 'c'
```

Example: toUpper

- Goal: Write toUpper( ), our own version of Matlab's upper( ), a function to convert a string to all uppercase
  - We want to do this without using Matlab's function upper( )
- Function header

```
function str = toUpper(str)
% Post: Convert string so all letters are upper case
% Pre:  Input is a string
```

  - Post = What is supposed to have happened when function is done (i.e., what the function does)
  - Pre = What assumptions are being made when function starts

Converting to Uppercase

- Idea: 'A' – 'a' has the same value as 'B' – 'b' which has the same value as 'C' – 'c', etc.
  - All we have to do is add the right number to a lowercase letter and we'll have the equivalent uppercase letter

```
>> char('a' + ('A' - 'a'))

ans =
A

>> char('e' + ('A' - 'a'))

ans =
E
```

toUpper.m

```
function str = toUpper(str)
% Post: Convert string so all letters are upper case
% Pre:  Input is a string
% This function is not really necessary since upper()
% does the same thing

diff = 'A' - 'a';
for k = 1:length(str)   % Check each letter
   if 'a' <= str(k) && str(k) <= 'z'
      str(k) = char(str(k) + diff);
   end
end
```

Example: Capitalize First Letters

- Goal:
  - Write a function to capitalize just the first letter of each word in a string
  - Assume the string consists entirely of letters and spaces
- Function header

```
function result = capitalize(str)
% Post: Convert string so each word has just first letter capitalized
% Pre:  Input string consists entirely of letters & spaces
```

What's Wrong with This Version?

```
function str = capitalize(str)
% Post: Convert string so each word has just first letter capitalized
% Pre:  Input string consists entirely of letters & spaces

str = lower(str);        % Make sure all letters are lowercase
for k = 1:length(str)    % Check each letter
   if isspace(str(k-1)) && isletter(str(k))
      str(k) = upper(str(k));
   end
end
```

```
>> capitalize('hello there what is this')
??? Attempted to access str(0); index must be a positive integer or logical.
Error in ==> capitalize at 7
   if isspace(str(k-1)) && isletter(str(k))
```

capitalize.m

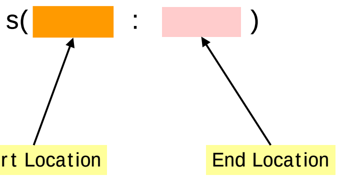
```
function str = capitalize(str)
% Post: Convert string so each word has just first letter capitalized
% Pre:  Input string consists entirely of letters & spaces

str = lower(str);       % Make sure all letters are lowercase
if isletter(str(1))     % Check for an initial letter
   str(1) = upper(str(1));
end
for k = 2:length(str)   % Check each remaining letter
   if isspace(str(k-1)) && isletter(str(k))
      str(k) = upper(str(k));
   end
end
```

```
>> capitalize('hello there what is this')
ans =
Hello There What Is This
```

Extracting Substrings

```
s = ‘abcdef’;

x = s(3)                % x = ‘c’
x = s(2:4)              % x = ‘bcd’
x = s(length(s))        % x = ‘f’
```

Colon Notation



s(:)

Start Location

End Location

Using the Word "end"

- In Matlab, the work "end" is *overloaded*
  - Used to terminate an if-statement
  - Used to terminate a for-statement
  - Used to terminate a while-statement
  - Used to represent the *last index* of a vector

```
s = 'abcdef';

x = s(end)                % x = 'f'
y = s(3:end)              % y = 'cdef'
```

Replacing Substrings

```
s = 'abcde';

s(2:4) = 'xyz'              % s = 'axyze'


s = 'abcde';

s(2:4) = 'wxyz'             % Error
```

- Dimensions must match

What is the final value of s?

```
s = 'abcde';
for k=1:3
        s = [ s(4:5) s(1:3)];
end
```

A. abcde
B. bcdea
C. eabcd
D. deabc

What gets printed?

```
t = 5;
b = myF(t);
fprintf('%d', t);
```

```
function y = myF(x)
t = 2 + x;
y = 2 * t;
```

A: 7
B: 6
C: 5
D: ERROR (t is undefined)

What happens when these statements are executed?

```
A = [3 4]
A = [A'  ones(2,1)]
A = [A A A]
```

A. Error in 2nd statement
B. Error in 3rd statement
C. In the end, A is a 2-by-6 matrix
D. In the end, A is a 6-by-2 matrix
E. In the end, A is a vector of length 3

How many X's are printed?

```
for k = 9:1
    disp('X')
end
```

A. 10
B. 9
C. 8
D. None; an error is reported
E. None; no error is reported

Many Operators Work on Entire Vectors

- Most Matlab operators are designed to work on entire vectors or entire matrices
  - This includes arithmetic, relational, and logical operators
  - Also includes most built-in functions (e.g., sin, cos, mod, floor, exp, log, etc.)
- Code that operates on entire vectors (or matrices) instead of on scalars is sometimes called *vectorized code*
- Examples

```
x = [10 20 30];
y = 1:3;
z = [2 1 2];

% Addition, subtraction
x + y        % [11 22 33]
x – y        % [9 18 27]

% Mult, division, power
% Must include the DOT “.”
x .* y       % [10 40 90]
x ./ y       % [10 10 10]
x .^ z       % [100 20 900]
```

Dot-Operators

- Matlab is especially set up for Linear Algebra
  - Thus, “*”, “/”, and “^” correspond to matrix operations
- Term-by-term operators use “.*”, “./”, and “.^”
  - Matlab documentation calls these “array operations” (as opposed to “matrix operations”)
- Why doesn’t Matlab include operators “.+” and “.-”?

Shapes Must Match

- Examples

```
a = [4 8 12]
b = [1; 2; 4]  % Column vector

a + b          % Error
a + b'         % [5 10 16]

a ./ b         % Error
a' ./ b        % [4; 4; 3]
```

- Exception to shape matching
  - Scalars follow special rules
  - “A scalar can operate into anything”
- Scalar examples

```
a + 1        % [5 9 13]
10 + a       % [14 18 22]
2 .* a       % [8 16 24]
a ./ 2       % [2 4 6]
24 ./ a      % [6 3 2]
a .^ 2       % [16 64 144]
```

Example: Pair-Sums

- Given a vector, report the vector of pair-sums (i.e., the sums of adjacent items)
  - Example: The pair-sum for [7 0 5 2] is [7 5 7]
- Function header

```
function s = pairSum(v)
% Return vector v's pair sums
```

- Iterative code

```
function s = pairSum(v)
% Return vector v's pair sums
s = [];
for k = 1: length(v)-1
   s(k) = v(k) + v(k+1);
end
```

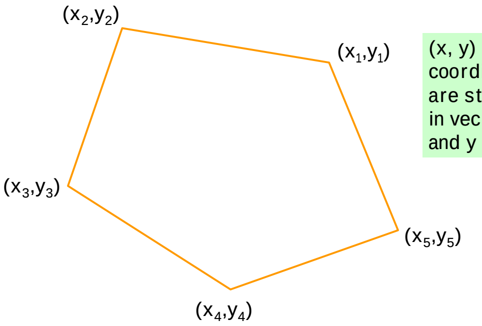
- Vectorized code

```
function s = pairSum(v)
% Return vector v's pair sums
s = v(1:end-1) + v(2:end);
```

Playing with Polygons

Playing with Functions that use Vectors

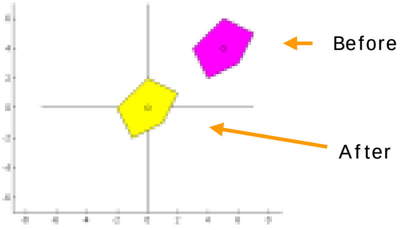
A Polygon



(x, y) coordinates are stored in vectors x and y

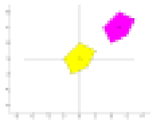
Operation 1: Centralize

- Move a polygon so that its center (the centroid of its vertices) is at the origin



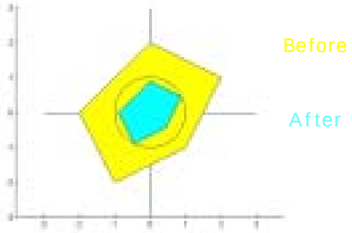
Centralize.m

```
 function [xNew,yNew] = Centralize(x,y)

  n = length(x);

% Compute the centroid...
  xBar = sum(x)/n; yBar = sum(y)/n;

% Translate the polygon...
  xNew = x-xBar; yNew = y-yBar;
```



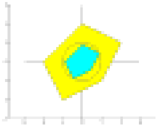
Operation 2: Normalize

- Shrink (or enlarge) the polygon so that the vertex furthest from the origin is on the unit circle



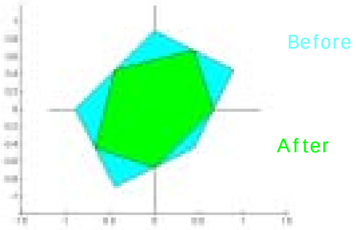
Normalize.m

```
 function [xNew,yNew] = Normalize(x,y)

% Max distance to origin...
  d = max(sqrt(x.^2 + y.^2));

% Normalize so furthest vertex is on the unit circle..
  xNew = x/d; yNew = y/d;
```



Operation 3: Smooth

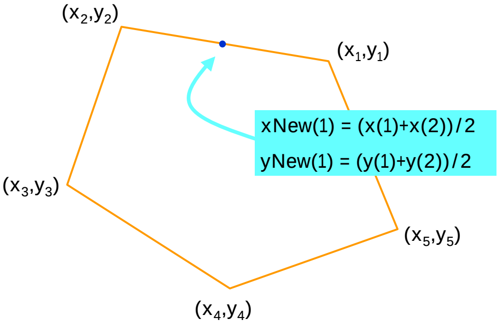
- Create a new polygon by connecting the midpoints of the polygon edges



Idea for Smooth

```
function [xNew,yNew] = Smooth(x,y)
n = length(x);
xNew = zeros(n,1);
yNew = zeros(n,1);
for i=1:n
        Compute the midpoint of $i^{th}$ edge
        Store in xNew(i) and yNew(i)
end
```

Computing the Midpoint



```
xNew(1) = (x(1)+x(2))/2
yNew(1) = (y(1)+y(2))/2
```

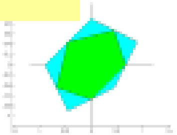
Code for Smooth

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

- Results in a subscript out of bounds error when k is n

Smooth.m

```
function [xNew,yNew] = Smooth(x,y)

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



Proposed Simulation

Create a polygon with randomly located vertices

Repeat:
    Centralize
    Normalize
    Smooth