



## Vectorized Code

Lecture 12 (Mar 2)  
CS100M - Spring 2006

## Announcements

- Project 3
  - Is either online now or will be online later today
  - Due: Thursday, March 9

## Topics

- Reading: CFile 9, Section 9.2
- Recall
  - Matlab vectors (1D arrays)
  - Characters & Strings
  - Matrices (2D arrays)
- Plans for today
  - Vectorized code
  - Pre-allocating arrays
  - Logical arrays

## Vectorized Code

- Most Matlab operations are designed to work on entire vectors or entire matrices
  - This includes arithmetic, relational, and logical operations
  - 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 said to be *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);
```

## Relational Operators

- Comparison operators (e.g., "<", ">", "==", etc.) also operate term-by-term, creating arrays of boolean values
- Examples
 

```
a = [7 0 5 2 4 6]
b = 1:6
a < b      % [0 1 0 1 1 0]
a == b     % [0 0 0 0 0 1]
```

## Logical Operators

- Logical operators (e.g., "&", "|") also operate term-by-term, creating arrays of boolean values
  - In Matlab, any nonzero value is considered to be "true"
- Examples
 

```
a = [7 0 5 2 4 6]
b = 1:6
a & b      % [1 0 1 1 1 1]
a < b & mod(b,2) == 0 % [0 1 0 1 0 0]
a < b && mod(b,2) == 0 % Error
```

## Short-Circuit Logical Operators

- Why two versions (&, &&) of "and"?
  - In <operand> & <operand>, both operands are evaluated before the &-operation is done
  - In <operand> && <operand>, the first operand is evaluated; if it's false then we don't bother evaluating the other operand
- Similar for the two versions (|, ||) of "or"
  - In <operand> || <operand>, the first operand is evaluated; if it's true then we don't bother evaluating the other operand
- Example use:
 

```
while (k > 0 && v(k) < 100) % Without short-circuit, Error
    ...
```

## Example: How Many F's?

- Goal: Determine how many times a particular character appears in a string
  - Example: How many f's in "An example of efficiently finding f"
- Function header
 

```
function n = charCount(s,c)
% Report # of c's in string s
```
- Iterative code
 

```
function n = charCount(s,c)
% Report # of c's in string s
n = 0;
for k = 1: length(s)
    if s(k) == c
        n = n + 1;
    end
end
```
- Vectorized code
 

```
function n = charCount(s,c)
% Report # of c's in string s
n = sum(c == s);
```

## Pre-allocating Arrays

- Recall the iterative version of the pair-sum example
 

```
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
```
- It will run faster if we *pre-allocate* the array *s*

```
function s = pairSum(v)
% Return vector v's pair sums
s = zeros(length(v) - 1);
for k = 1: length(v)-1
    s(k) = v(k) + v(k+1);
end
```
- Vector *s* grows as needed
  - This works fine in Matlab, but...
  - It's slow
- Note though that vectorized code is even faster!

## Improving Efficiency

- For efficiency
  - Use vectorized code if possible
  - If you must use a loop, pre-allocate any arrays
- We can write a program to test these ideas
  - Matlab provides built-in functions "tic" (start timer) and "toc" (report time elapsed since tic)

## Example: Polynomial Evaluation

```
function p = polyEval(coeff, x)
% Evaluate polynomial at x; coeff is vector of coefficients.
% coeff(1) is the constant term.
```

% Original code

```
p = 0;
for k = 1:length(coeff)
    p = p + coeff(k)*x^(k-1);
end
```

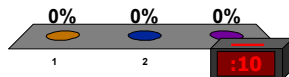
% Vectorized replacement code

```
d = length(coeff) - 1;      % Degree of polynomial
p = sum(coeff .* (x .^ (0:d)))
```

Which will produce a vector of perfect squares up to 100?

1. `(1:10) .^ 2`
2. `(1:10) .* (1:10)`
- ➔ 3. Both of the above

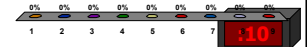
0 of 400



Which will produce the vector of even numbers between 1 and 101?

1. `1:2:101`
2. `2:2:101`
3. `2:2:100`
4. `2 .* (1:50)`
5. `(1:50) .* 2`
6. All of the above
- ➔ 7. All of 1 thru 5 except 1
8. All of 1 thru 5 except 3
9. All of 1 thru 5 except 5

0 of 400

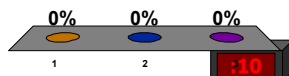


What does this code do?

`c = sum(mod(v, 2) == 0)`     % v is a vector

1. Nothing; there is an error
- ➔ 2. c is the number of even numbers in vector v
3. c is the number of odd numbers in vector v

0 of 400

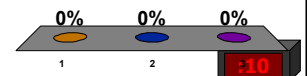


What does this code do?

`c = floor(sqrt(v)) .^ 2`     % v is a vector

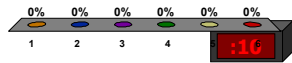
1. Nothing; there is an error
2. c is the number perfect squares in vector v
- ➔ 3. Each number in v is converted into a nearby perfect square

0 of 400



Which of the following will *not* produce a 3-by-3 matrix of 3's?

1. `3.*ones(3,3)`
2. `2+ones(3,3)`
3. `zeros(3,3)+3`
4. `[3 3 3; 3 3 3; 3 3 3]`
5. `z = ones(3,3); z(:,:) = 3`
6. None of the above; they all work



## Neighborhood of a Cell

- We define the *neighborhood of a cell* to be the cell itself and all adjacent cells (including diagonally adjacent)

7	0	7	0	5
2	4	5	2	6
4	6	3	8	1
7	0	5	2	4
3	8	6	2	1

The neighborhood of cell(2,4)

The neighborhood of cell(5,2)

## Min of a Neighborhood

- Goal:**  
Write a function `minInNeighborhood(M, row, col)` that reports the minimum value in neighborhood of cell(row, col) in matrix `M`
- Function header**  
Function val = `minInNeighborhood(M, row, col)`  
% Return min in neighborhood of (row, col) in M

## Ask Yourself Questions

- Do we know how to solve a similar problem?
  - Yes, we already have code to find the min of a matrix
- Can we make a neighborhood into a matrix?
  - Yes, Matlab makes it easy to do submatrices
  - Neighborhood of `M(row, col)` is `M(row-1:row+1, col-1:col+1)`
- What happens near the edges?
  - Doesn't work near the edges: we "fall off"
- What can we do to fix up the edges?
  - We can make the code more complicated, or...
  - We can modify the matrix so we *can't* fall off
- If we add a border around `M`, what goes in the border?
  - `realmax`