



UNIVERSITY
OF WOLLONGONG
IN DUBAI

Lecture 7 Week 8

Mohamad Nassereddine



UNIVERSITY
OF WOLLONGONG
IN DUBAI

Vectorisation

Vectorisation is the process of revising loop-based, scalar-oriented code to use MATLAB array operations

There many advantages to vectorising code:

Appearance: easier to understand.

Less Error Prone: code without loops is generally shorter - fewer opportunities for errors.

Performance: vectorised functions are highly optimised and generally faster

Disadvantages

Many of these functions/functionality have no equivalent in the likes of C/C++, Java, VBA, etc, making the transition to other languages harder if you don't understand loops, etc.



Vectorisation

As a result, many of the in-built MATLAB function are optimised for arrays and matrices
You have already been using some vectorised functions

normal multiplication

Array version of multiplication

Loops-based Solution	Vectorised Solution
<pre>% 2 times array a = 1:10; for i = 1:10 b(i) = 2 * i; end</pre>	<pre>% 2 times array a = 1:10; b = a .* 2;</pre>
<pre>% calc sum s = 0; for i = 1:10 s = s + i; end</pre>	<pre>% calc sum s = sum(1:10);</pre>
<pre>% mult table for i=1:10 for j=1:10 m1(i,j) = i*j; end end</pre>	<pre>% mult table [i,j] = meshgrid(1:10); m2 = i .* j;</pre>

Vectorised functions

Always check whether a function already exists that performs the action you want on an array.

For example, before starting to write a script to flip an array left-to-right, check for existing functions

```
% code snippet to flip array
a = 1:10;
len = length(a);
halflen = floor(len/2);
for i=1:halflen
    % swap (i) and (length+1-i)
    temp = a(i);
    a(i) = a(len+1-i);
    a(len+1-i) = temp;
end
```

```
Command Window

>> a = 1:10
a =
     1     2     3     4     5     6     7     8     9    10

>> fliplr(a)
ans =
    10     9     8     7     6     5     4     3     2     1

fx >> |
```

Vectorising loops

Take this loop-based example:

Each element in the array is assigned the value of the previous element plus i^2

It might not immediately be obvious how to vectorise this program

We can use the fact that vectors can be indexed by other vectors and
Use pre-existing functions

```
% size of array
n = 10;
% first element =1
x(1) = 1;
% update each element
for i = 2:n,
    x(i) = x(i-1) + i^2;
end
```

```
% size of array
n = 10;
% first element =1
x(1) = 1;
% indices excluding 1st element
i = 2:n;
% save elements with the non-
% previous part of formula only
x(i) = i.^2;
% use vectorised cumulative sum
x = cumsum(x);
```

Logical vectors

Vectorisation can also include use of logical vectors.

Logical vectors are an array of logical values that refer to a condition as true or false for each element in the array

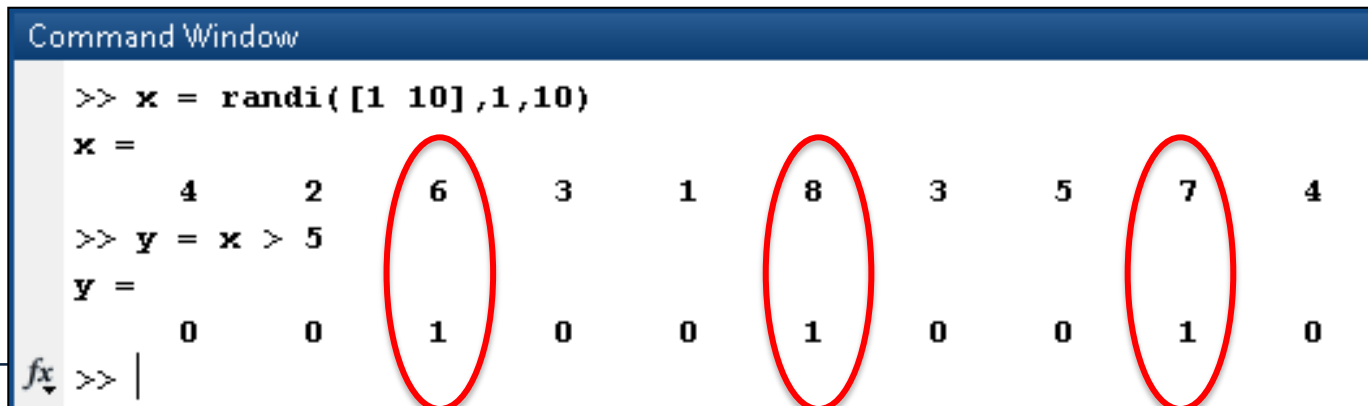
In the example below, a random 1x10 array of numbers is generated (values between 1-10)

When the conditional operator ' $>$ ' is called, it creates a logical vector with **true/false** values for each index

Only in the locations where $\mathbf{x(i)} > 5$, will a **1** appear (**true**)

These vectors are sometimes called "*masks*"

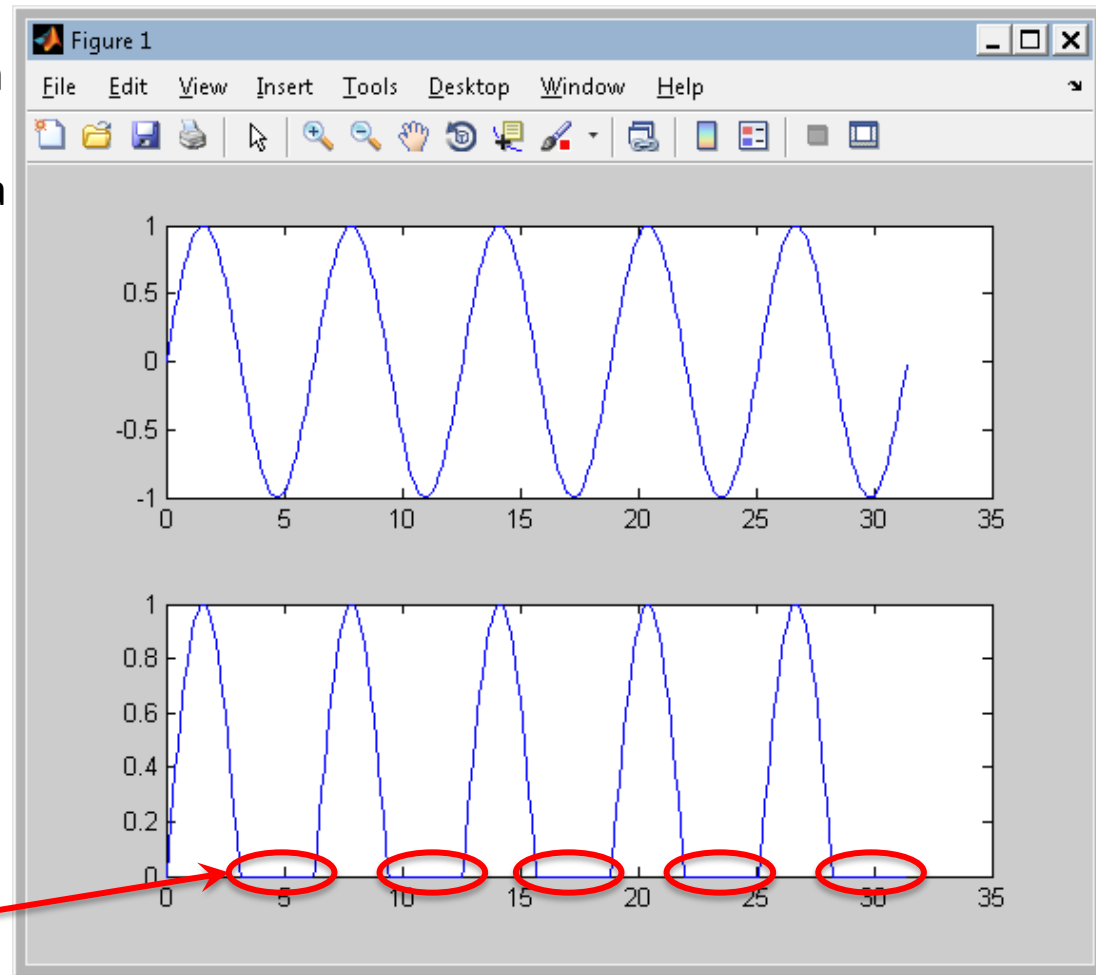
```
Command Window
>> x = randi([1 10],1,10)
x =
     4     2     6     3     1     8     3     5     7     4
>> y = x > 5
y =
     0     0     1     0     0     1     0     0     1     0
```



Logical Vectors example

We can use logical vectors to place a “mask” in front of a series of data in order to include or exclude that data we want.

```
% time series to 10*pi  
t = 0:0.1:10*pi;  
  
% normal sin(t)  
a = sin(t);  
plot(t, a);  
  
% sin(t) without negatives  
b = (a > 0) .* a;  
plot(t, b);
```



Searching algorithms

In the era of “**Big Data**”, search algorithms have seen a lot of publicity

Google’s PageRank algorithm ranks websites according to the number of links from other high rank sites

Google last made changes to its mobile search algorithm to give “mobile-friendly” pages higher rank (April, 2015)

But where did search algorithms start? Let’s start by searching for a number in an array

Linear Search

Binary Search



Linear Search

The basic linear search starts at array element **a(1)**, and iterates through every index in the array to find the value **n**

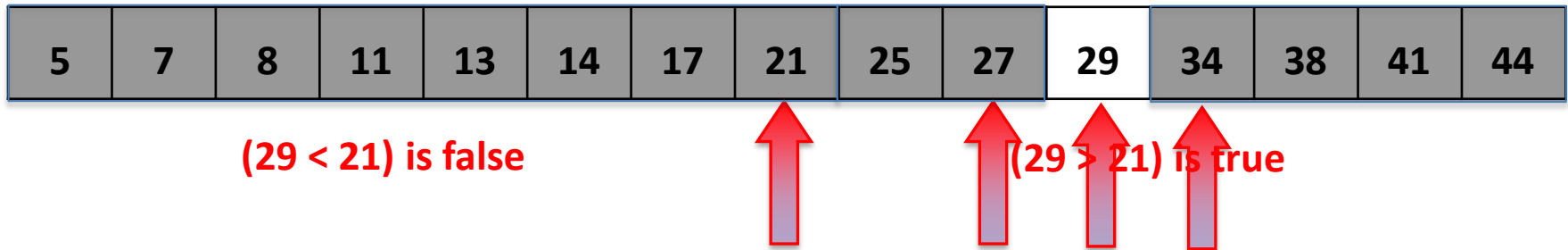
```
% Linear Search of array a to find n
found = 0;
for i=1:length(a)
    if a(i) == n
        found = i;
        break;
    end
end
% output whether found or not
if found==0
    disp('not found');
else
    disp(['found at:', num2str(i)]);
end
```



Binary Search

Binary Search is quicker, but assumes the array is already sorted in-place
Starts at the halfway point, then narrows down the search by choosing which half to continue searching

For example: let's search for the number 29



Sorting

Sorting is required in many applications
Multiple algorithms exist to sort effectively
The simplest to understand is **bubblesort**

Command Window

```
>> a = randi(100,1,10)
```

```
a =
```

98

65

81

46

44

83

9

14

18

40

fx

```
>>
```

Command Window

```
>> sorted = sort(a)
```

```
sorted =
```

9

14

18

40

44

46

65

81

83

98

fx

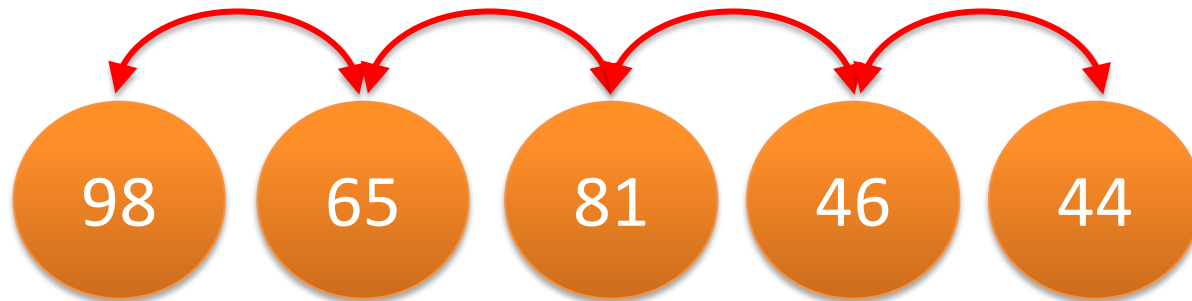
```
>>
```

Bubble Sort

Take an array and compare consecutive values.

Swap if the left value is larger than the right value

There will be multiple passes required through the array. Let's start with an example:



A First Step

As with any larger problem, it is worthwhile approaching with an initial first step
Let's take a look at the first two values - and swap them if needed

```
%% "a" is the array to be sorted
%
% Step 1 - check if a(1) and a(2) are sorted
% if not, swap them - a(1) <=> a(2)
if a(1) > a(2)
    temp = a(1); % temporarily save a(1)
    a(1) = a(2); % overwrite a(1) with a(2)
    a(2) = temp; % overwrite a(2) with temp
end % no need for an "else" part
```

a_before =	98	65	81	46	44	83	9	14	18	40
a_after =	65	98	81	46	44	83	9	14	18	40

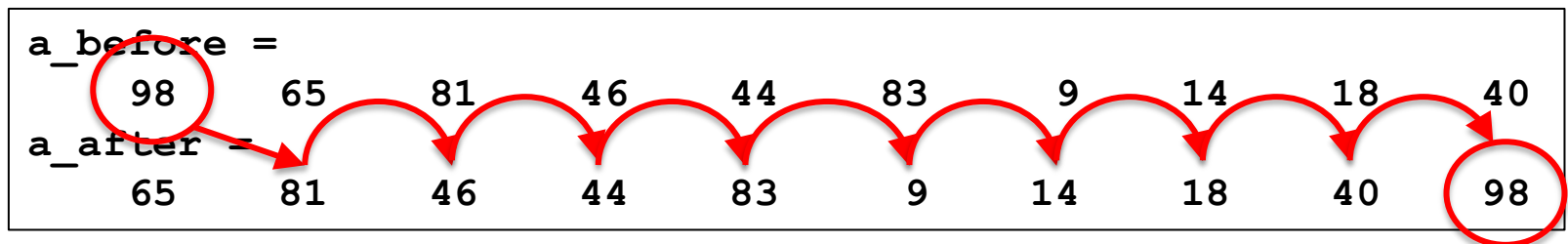


A Second Step

But that only sorts the first two values.

Let's apply this to every pair of values in the array

```
%% "a" is the array to be sorted
%
% Step 2 - compare each a(i) with a(i+1)
% if not sorted, swap - a(i) <=> a(i+1)
for i = 1:9 % There are 9 pairs
    if a(i) > a(i+1)
        temp = a(i); % Perform swap
        a(i) = a(i+1); % between a(i)
        a(i+1) = temp; % and a(i+1)
    end
end
```



A Third Step

Because the largest number is now in place, on the far right, we want to repeat the same process on array **a (1 : 9)** , then again on **a (1 : 8)** , then again on **a (1 : 7)** ... until we are certain that all parts of the array are sorted

We need to place our original loop inside another loop, however each subsequent loop internal loop can be 1 smaller than the previous



A Third Step

Include an outer for-loop that sets the upper boundary of the inner loop to be one less, each time

```
%% "a" is the array to be sorted
% Step 3 - Repeat step 2 for n = 9->1
for n = 9:-1:1
    for i = 1:n % There are "n" pairs
        if a(i) > a(i+1)
            temp = a(i); % Perform swap
            a(i) = a(i+1); % between a(i)
            a(i+1) = temp; % and a(i+1)
        end
    end
end
```

a_before =	98	65	81	46	44	83	9	14	18	40
a_after =	9	14	18	40	44	46	65	81	83	98



Improvements?

This code will only work for arrays that are exactly 10 elements long

We could use **length(a)** to determine the size of the array before commencing our bubble sort program

We could use an indeterminate looping system for the outer loop (**while**)

This would eliminate additional passes through the array if not required

This is the basis for the bubblesort in the textbook



After n=9:

65 81 46 44 83 9 14 18 40 98

After n=8:

65 46 44 81 9 14 18 40 83 98

After n=7:

46 44 65 9 14 18 40 81 83 98

After n=6:

44 46 9 14 18 40 65 81 83 98

After n=5:

44 9 14 18 40 46 65 81 83 98

After n=4:

9 14 18 40 44 46 65 81 83 98

After n=3:

9 14 18 40 44 46 65 81 83 98

After n=2:

9 14 18 40 44 46 65 81 83 98

After n=1:

9 14 18 40 44 46 65 81 83 98



Other sorting algorithms

Insertion sort

Builds up the sorted array by “inserting” next value

Selection sort

Select each item as you need it in the array

Heap sort also uses some aspects of selection sort

Merge sort

Divide the array into sub-arrays, sort them and merge

Quick Sort

Partition the array at pivot point. Recursively sort.

This is the algorithm used by MATLAB's sort



Some infographics...

http://en.wikipedia.org/wiki/Selection_sort

<http://en.wikipedia.org/wiki/Heapsort>

http://en.wikipedia.org/wiki/Merge_sort

<http://en.wikipedia.org/wiki/Quicksort>

http://en.wikipedia.org/wiki/Insertion_sort



Recursion

- Recursion is a divide-and-conquer approach
- A technique used to solve problems based on recurrence relations
- A function is able to call itself with a smaller input
- **Fast sorting** is a real-world example that uses recursion
- Another example: performing an operation on all files in a folder including sub-folders
 - You can recurse through the directory structure, calling the function for each new sub-folder



HOW TO CREATE RECURSIVE FUNCTIONS IN MATLAB

- You create a function that keeps calling itself until a condition is satisfied, and then the function delivers an answer based on the results of all those calls.
- This process of the function calling itself multiple times is known as *recursion*, and a function that implements it is a *recursive function*.
- The most common recursion example is calculating **factorial (n!)**, where n is a positive number.
- (Calculating a factorial means multiplying the number by each number below it in the hierarchy. For example, **4!** is equal to **4*3*2*1** or 24.)

```
function y = fact(n)
% FACT Recursive
definition of n!
if n > 1
    y = n * fact(n-1);
else
    y = 1;
end;
```



Recursive function

- A recursive function must always have an **ending point** — a condition under which it won't call itself again. In this case, the ending point is the **else** clause.
- When y is finally less than 1, y is assigned a value of 1 and simply returns, without calling **fact ()** again.

```
function y = fact(n)
% FACT Recursive definition of n!
if n > 1
    y = n * fact(n-1);
    disp(['output for iteration',
        num2str(n), ' = ', num2str(y)]);
else
    y = 1;
    disp(['output for iteration',
        num2str(n), ' = ', num2str(y)]);
end;
```

```
>> fact(4)
output for iteration 1 = 1
output for iteration 2 = 2
output for iteration 3 = 6
output for iteration 4 = 24

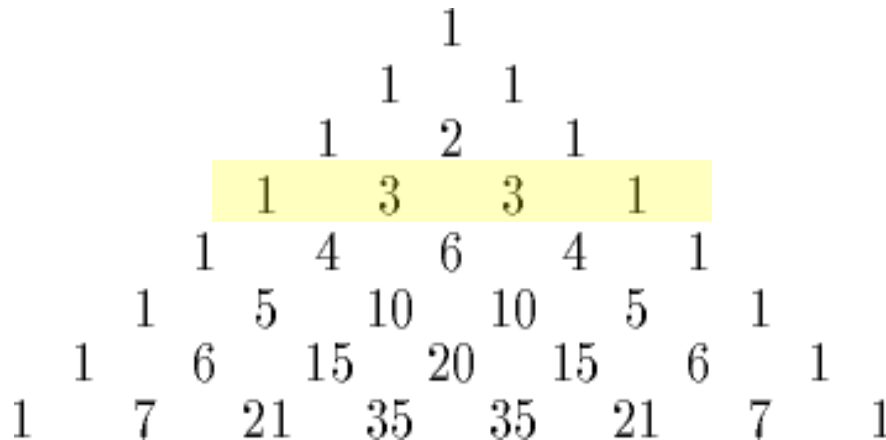
ans = 24
```



The Pascal's triangle is a

- For example, the fourth row in the triangle shows numbers 1 3 3 1, and that means the expansion of a cubic binomial, which has four terms.

- $(x + y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$




```
function [ y ] = Pasc(n,k)
```

```
%Let n be the row (starting from row 0) and k be the column  
(starting from column 0)
```

```
if n == 0 || k == 0 || n == k y=1;  
else  
    y=Pasc(n-1,k-1)+Pasc(n-1,k);  
end
```

									Row 0		
								1	Row 1		
							1		1	Row 2	
						1		2		1	Row 3
				1		3		3		1	Row 4
		1		4		6		4		1	Row 4

- Row 0 Column 0 if statement gives $y=1$ ($n=0, k=0$)
- Row 1 Col 0 if statement sets $y=1$ ($k=0$)
- Row 1 Col 1 if statement sets $y=1$ ($n=k$)
- Row 2 Col 0 if statement sets $y=1$
- Row 2 Col 2 if statement sets $y=2$

Calculate Row 2 Col 1 = Row 1 Col 0 + Row 1 Col 1 = $(1+1)$

- Row 3 Col 0 = Row 3 Col 3 = 1 if statement

Calculate Row 3 Col 1 = Row 2 Col 0 + Row 2 Col 1 = $1+2 = 3$;

Row 3 Col 2 = Row 2 Col 1 + Row 2 Col 2 = $2+1 = 3$

Row 4 Col 0 = Row 4 Col 4 = 1 if statement

Calculate Row 4 Col 1 = Row 3 Col 0 + Row 3 Col 1 = $1+3 = 4$

Row 4 Col 2 = Row 3 Col 1 + Row 3 Col 2 = $3+3 = 6$

Row 4 Col 3 = Row 3 Col 2 + Row 3 Col 3 = $3+1 = 4$