

# MATLAB for SMS

## Lesson 1

### Array Operations

June 6, 2011

course notes by Nick Taylor

# Array Fundamentals

Scalars and Vectors

Regularly Spaced Vectors and a Simple Matrix

Dimensions and `size`, `zeros` & `ones`

Indexing

Accessing Elements

# Array Fundamentals: Scalars and Vectors

- Use of the term scalar describes a constant real number unless otherwise noted.
- Use of the term vector describes a discrete, 1-dimensional array of scalars or some other data type. A scalar should be interpreted as a vector with a single element.
- Often a vector's contents will be unimportant to the operation being described. In these instances, each discrete element of the array will be displayed as a shape. Something like: ●
- The most basic way to construct a row vector is to use the square brackets. A space or a comma between each element to indicate a horizontal array.

```
>> [1 2 3 4]
```

ans =

	1	2	3	4
	●	●	●	●

- For a column vector we use the *semicolon operator* to indicate we want a new row to begin.

```
>> [1;2;3;4]
```

ans =

1	●
2	●
3	●
4	●

# Array Fundamentals: Regularly Spaced Vectors and a Simple Matrix

- The *colon operator* is useful in constructing regularly spaced vectors:

```
>> 1:4
```

```
ans =
```

```
1      2      3      4
```

- The default interval is 1, but the interval can be specified as well:

```
>> 1:0.5:4
```

```
|
```

```
ans =
```

```
1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000
```

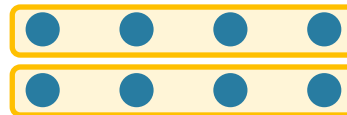
- A matrix is a 2-dimensional array. It possesses both *rows* and *columns*.

```
>> [1 2 3 4; 5 6 7 8]
```

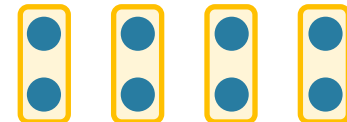
```
ans =
```

```
1      2      3      4|
5      6      7      8
```

rows



columns



Notice: don't confuse this *semicolon* ; with the one that suppresses output.

# Array Fundamentals: Dimensions and `size`, `zeros` & `ones`

- The `size` command returns a 2-element vector containing the numbers of rows and columns in the matrix. e.g., on the previous slide, we constructed a 2x4 matrix:

```
>> size([1 2 3 4; 5 6 7 8])
```

```
ans =
```

```
2     4
```



- Specifying a dimension, 1 for rows and 2 for columns, returns a scalar containing the specified dimension:

```
>> size([1 2 3 4; 5 6 7 8],1)
```

```
ans =
```

```
2
```

```
>> size([1 2 3 4; 5 6 7 8],2)
```

```
ans =
```

```
4
```

- MATLAB contains several special matrices, such as `zeros` and `ones`, that are constructed like this:

```
>> zeros(2,4)
```

```
ans =
```

```
0     0     0     0
0     0     0     0
```

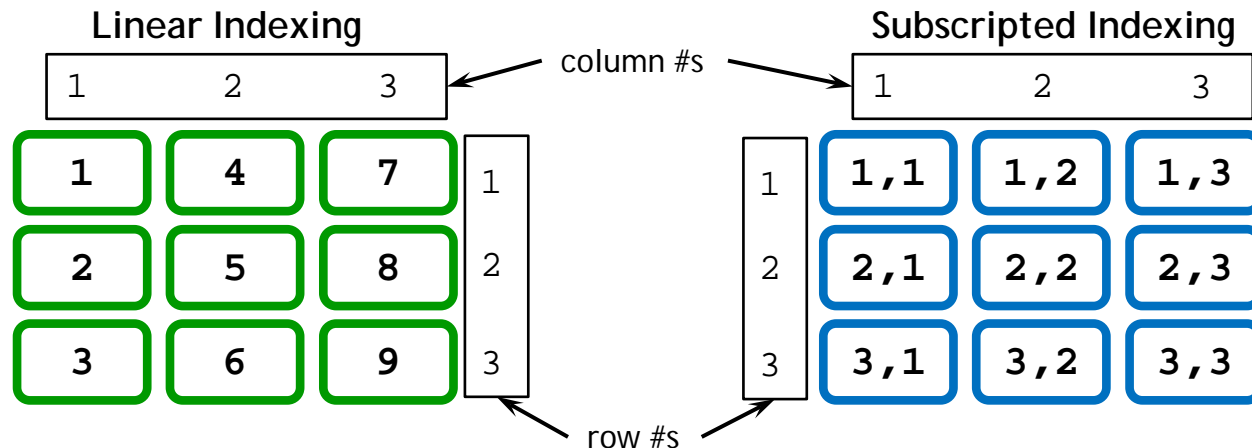
```
>> ones(2,4)
```

```
ans =
```

```
1     1     1     1
1     1     1     1
```

# Array Fundamentals: Indexing

- An element's index is its "address" in the array. Each element in an array can be found by either of 2 very different indexing methods.



- We navigate to specific elements in the matrix A by entering `A(elements)`, where `elements` is a numerical array containing the indices we wish to access.
- All indices, whether linear or subscripted, must be positive, nonzero integers.
- The two methods are interchangeable, but sometimes it may be simpler to use one over the other. When it's necessary to switch from one to the other, use:

```
>> index = sub2ind(size(A),row,col)      [row col] = ind2sub(size(A),index)
```

# Array Fundamentals: Accessing Elements

- Elements and sets of elements can be accessed by specifying either their linear or subscripted indices. The 2x2 matrix A:

$$>> A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix}$$

- An element may be accessed in several different ways. For a single element, we specify its index. Notice that the index `end` points to the highest linear index or to the last element in a row or column

$$>> A(3) = A(1,2) = A(1,end) = 3 = 1,2$$

$$>> A(end) = A(2,end) = A(4) = A(2,2) = 4 = 2,2$$

- Entire rows and columns, as well as sets of rows and columns, may be accessed as so:

$$>> A([1 \ 2]) = A(1:2,1) = A(1:end,1) = A(:,1) = \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1,1 \\ 2,1 \end{bmatrix}$$

$$>> A([1 \ 3]) = A(1,1:2) = A(1,1:end) = A(1,:) = \begin{bmatrix} 1 & 3 \end{bmatrix} = \begin{bmatrix} 1,1 & 1,2 \end{bmatrix}$$

# Math Fundamentals

Matrix Multiplication

Elementwise Operators

Using Elementwise Arithmetical Operators



# Math Fundamentals: Matrix Multiplication

- Matrix multiplication is of rare practical use in SMS so its inclusion here will be brief. Consult the help files or a linear algebra textbook if you'd like more detail.
- The matrix product exists if and only if the inner matrix dimensions of the two matrices are equivalent. e.g., only if the number of columns in the first array matches the number of rows in the second array.
- For an array  $C = A * B$ , where A and B are both 2x2, each element of the matrix product is obtained as so:

Diagram illustrating the first row of the matrix product  $C = A * B$ . The first row of  $C$  is highlighted with a yellow border. It is calculated as the dot product of the first row of  $A$  (highlighted with a yellow border) and the columns of  $B$  (each highlighted with a red border). The calculation is shown as:

$$\begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix} = \begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix} \times \begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix}$$

Diagram illustrating the second row of the matrix product  $C = A * B$ . The second row of  $C$  is highlighted with a yellow border. It is calculated as the dot product of the second row of  $A$  (highlighted with a yellow border) and the columns of  $B$  (each highlighted with a red border). The calculation is shown as:


$$\begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix} = \begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix} \times \begin{bmatrix} 1,1 & 1,2 \\ 2,1 & 2,2 \end{bmatrix}$$

# Math Fundamentals: Elementwise Operators

- To add or subtract a scalar from an array A, we use this syntax:

`>> A + 3 = (  + 3 ) (  + 3 ) (  + 3 )`       $A = \text{blue circle} \quad \text{green triangle} \quad \text{yellow square}$

- Likewise, to multiply or divide an array by a scalar, we use this syntax:

`>> 3*A = ( 3*  ) ( 3*  ) ( 3*  )`

- When we try to add two arrays however, we find they are required to be equal in size.

```
>> A + B
??? Error using ==> plus
Matrix dimensions must agree.
```

- Instead, If B were as it is to the right, addition of the two vectors A and B would perform this operation:

$B = \text{green triangle} \quad \text{yellow square} \quad \text{blue circle}$

$$\begin{array}{ccccc} \text{blue circle} & \text{green triangle} & \text{yellow square} & + & \text{green triangle} & \text{yellow square} & \text{blue circle} & = & \text{blue circle} + \text{green triangle} & \text{green triangle} + \text{yellow square} & \text{yellow square} + \text{blue circle} \\ A & & & & B & & & & A + B & & \end{array}$$

- Such operations, those that act separately on each element of an array are called elementwise operations. They are quite useful in that they are faster than a computational loop and because their notation is so compact. They work (basically) like this:

# Math Fundamentals: Using Elementwise Arithmetical Operators

- As we saw on the previous slide, addition or subtraction of two arrays requires the arrays to be of equivalent size.

$$A = \text{blue circle} \quad \text{green triangle} \quad \text{yellow square}$$

$$B = \text{green triangle} \quad \text{yellow square} \quad \text{blue circle}$$

- Elementwise multiplication and division are indicated by placing a dot (i.e., period.) immediately preceding the multiplication  $*$  or division  $/$  operator

$$>> A .* B = \text{blue circle} * \text{green triangle} \quad \text{green triangle} * \text{yellow square} \quad \text{yellow square} * \text{blue circle}$$

- Likewise, if we'd like to raise the elements of our vectors to some power:

$$A.^2 = \text{blue circle}^2 \quad \text{green triangle}^2 \quad \text{yellow square}^2$$

$$B.^(1/2) = \sqrt{\text{green triangle}} \quad \sqrt{\text{yellow square}} \quad \sqrt{\text{blue circle}}$$

- Notice that  $A.^(1/2)$  and `sqrt(A)` are equivalent expressions.

# Array Manipulation

Concatenation and Elimination

`transpose, fliplr, flipud`

`sortrows`

`sort`

`reshape`

`repmat`

# Array Manipulation: Concatenation and Elimination

Concatenation joins two matrices in the manner specified in the command.

A =  B = 

[A B] = 

- `size(A,1)` and `size(B,1)` must be equivalent

[A;B] = 

- `size(A,2)` and `size(B,2)` must be equivalent

[A B;B A] = 

- `size(A)` and `size(B)` must be equivalent in both dimensions

Elimination is identical to retrieval, except that the elements retrieved are deleted.

C = 

- Deleting part of a row or column will result in a row vector ordered by linear index.

C(3) = []; 

C([1 3]) = []; 

- if entire rows or columns are deleted, the remaining structure of the array is maintained.

C(1:2) = [];  
C(:,1) = [];  
C(1:end,1) = [];



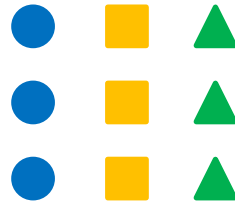
C(1,:) = [];  
C([1 3 5]) = [];  
C(1,1:end) = [];



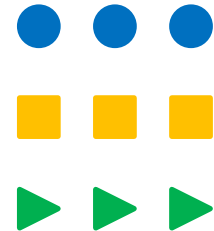
# Array Manipulation: transpose, flipplr, flipud

## transpose

- rows  $\longrightarrow$  columns
- columns  $\longrightarrow$  rows

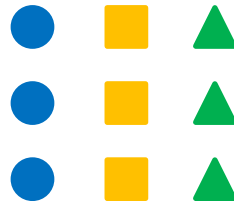


$A'$   
 $\xrightarrow{\text{transpose}(A)}$

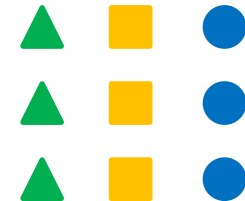


## flip horizontally

- rows maintained
- column order reversed

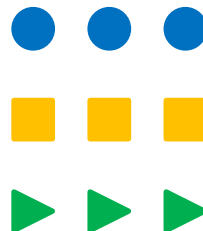


$\xrightarrow{\text{fliplr}(A)}$

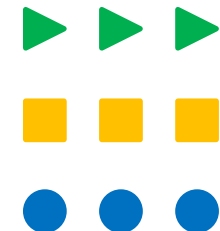


## flip vertically

- columns maintained
- row order reversed



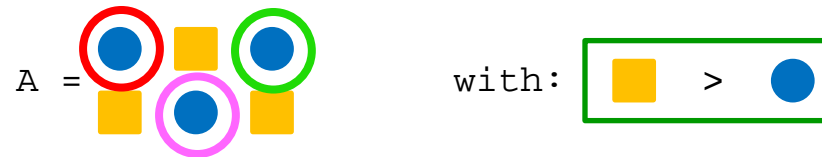
$\xrightarrow{\text{flipud}(A)}$



# Array Manipulation: `sortrows`

```
>> sortrows(A)
>> sortrows(A,column)
```

- `sortrows` sorts the rows of some array in ascending order (surprise!). It is most useful when an array must be sorted in one dimension while maintaining its order in the other dimension.
- the default column is 1. So, when called with no specified column, `sortrows` sorts based on the first column of the matrix.
- Sort by any column by specifying the column number in the command syntax.

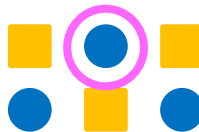


`sortrows(A) =`

`sortrows(A,1) =`




`sortrows(A,2) =`






`sortrows(A,3) =`



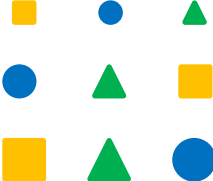

\*\* the max element of the column chosen in the command is marked with a color-coded 

# Array Manipulation: `sort`

```
>> sort(A)
>> sort(A,dim)
>> sort(A,dim)
>> sort(A,dim,'ascend')
>> sort(A,dim,'descend')
```

all  > all  > all 

- `sort` sorts (...) the elements of array in ascending or descending order. `sort` does not change the shape of the array, but be aware that elements will be ordered by increasing linear index without regard to their original locations.

A =       `sort(A, 'descend')` = 

- `sort` is straightforward if A is a vector:

A = 

`sort(A, 'descend')` = 

`sort(A, 'ascend')` = 

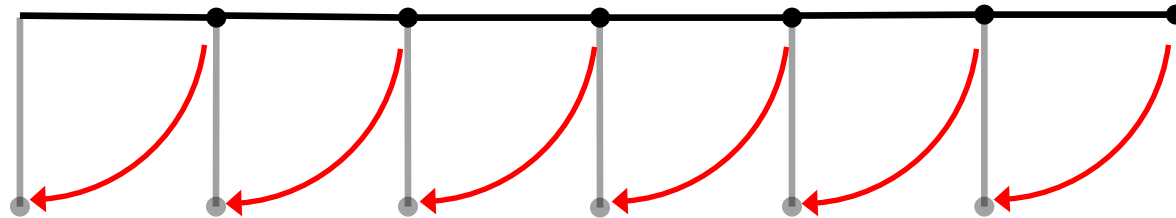


# Array Manipulation: `reshape`

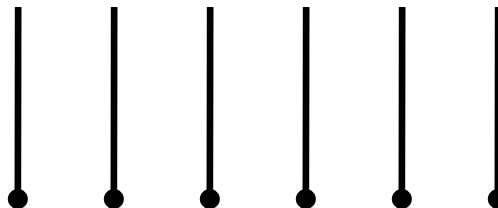
- `reshape(A,m,n)`
- Consider the horizontal line below to be a row vector  $A$  containing  $N$  elements. We would like to reshape it to be  $m \times n$ , where  $m$  and  $n$  are two arbitrary integers whose product is  $N$ .

$A =$  

- There are  $n$  segments of our vector, and each segment is  $m$  elements long. Using the command `reshape(A,m,n)`, we can visualize the reshaping process like this:



- This results in the  $m \times n$  matrix containing the same  $N$  elements as the original vector.



- If we are reshaping a matrix to a vector, we can simply reverse our visual logic.

## Array Manipulation: repmat

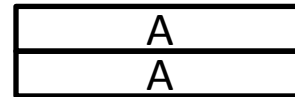
» repmat(A,m,n)



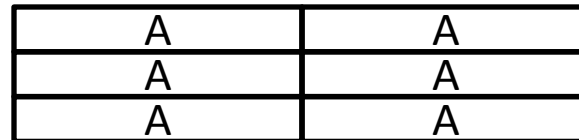
» repmat(A,1,2)



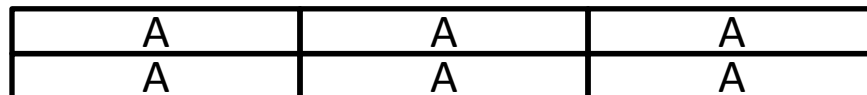
» repmat(A,2,1)



» repmat(A,3,2)



» repmat(A,2,3)



# Statistical Operators

Statistical Operators: `sum, mean, std, median, mode, var ...`

Statistical Operators: `max and min`

Statistical Operators: `max and min in 2-D`

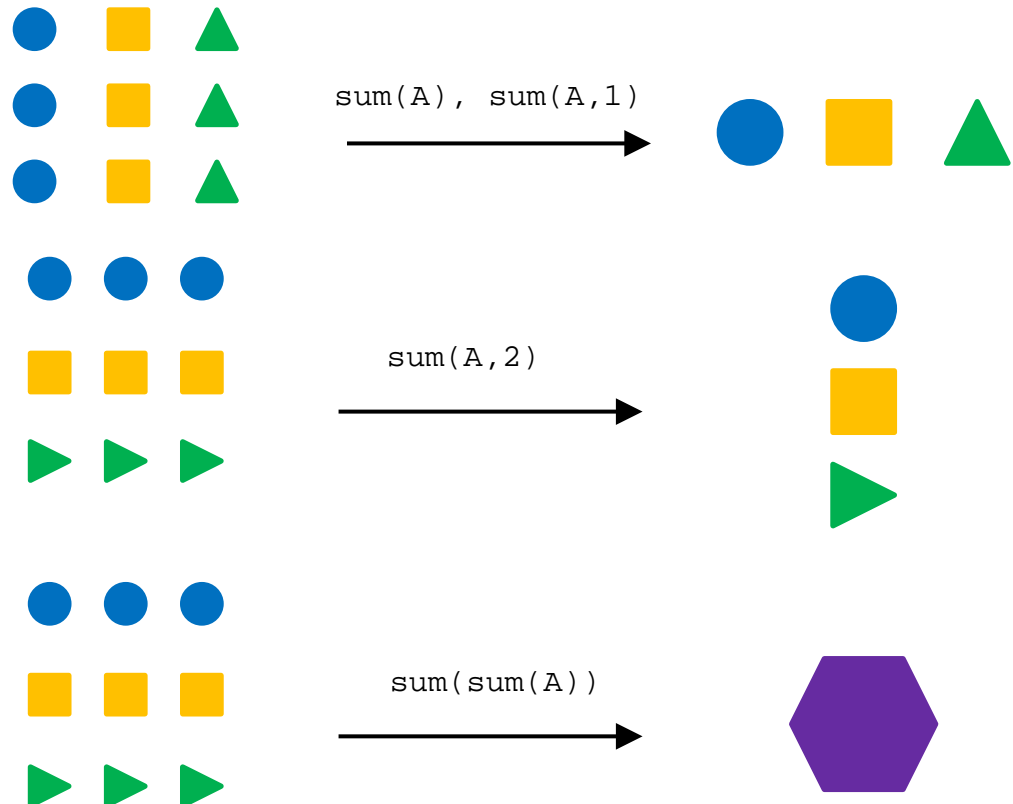
# Statistical Operators: sum, mean, std, median, mode, var ...

```
>> sum(A)
>> sum(A,dim)
```

- sum is one of a group of operators that return statistical quantities of the entire array.
- Conveniently, their options and their syntax is largely identical from function to function.

## Command Syntax

```
>> sum(A,dim)
>> mean(A,dim)
>> std(A,dim)
>> median(A,dim)
>> mode(A,dim)
>> var(A,dim)
...
```



# Statistical Operators: max and min

```
>> max(A), max(A,B,dim)
>> [mag,index] = max(A,B,dim)
```

all  > all  > all 

A =    


B =    

- In 1-D, max and min return a scalar that is the extreme value in the array.

max(A) = max(B) = 


- [mag,index] will return the maximum value, mag, as well as its linear index, index.

[mag,index] = max(A);

mag = 

index = 1

[mag,index] = max(B);

mag = 

index = 3

- max and min will also find the extrema of 2 equally sized arrays with max(A,B) is used. B can be a scalar here, e.g. max(A,0)

max(A,B) = max([A;B]) = max(  
     
    ) =    

# Statistical Operators: max and min in 2-D

- In 2-D, `max(A)` returns a row vector (like `sum`). Each element contains the max along each column.

$$\text{max}(C) = \text{yellow square} \quad \text{green triangle} \quad \text{blue circle}$$

- `max(max(A))` returns the absolute max of the entire array.

$$\text{max}(\text{max}(C)) = \text{yellow square}$$

$\text{all } \text{yellow square} > \text{all } \text{green triangle} > \text{all } \text{blue circle}$

$$C = \begin{bmatrix} \text{yellow square} & \text{green triangle} & \text{blue circle} \\ \text{yellow square} & \text{green triangle} & \text{blue circle} \\ \text{yellow square} & \text{green triangle} & \text{blue circle} \end{bmatrix}$$

- Notice the empty brackets when specifying dimensions. The default dim is 1.

$$\text{max}(C, [], 1) = \text{max}(C) = \text{yellow square} \quad \text{green triangle} \quad \text{blue circle} \quad \text{max}(C, [], 2) = \begin{bmatrix} \text{yellow square} \\ \text{yellow square} \\ \text{yellow square} \end{bmatrix}$$

- `[mag1, index1]` returns linear indices of the max while treating the row or column that the max

$$\begin{aligned} [\text{mag1}, \text{index1}] &= \text{max}(C); & [\text{mag2}, \text{index2}] &= \text{max}(A, [], 2); \\ \text{mag1} &= \text{yellow square} \quad \text{green triangle} \quad \text{blue circle} & \text{mag2} &= \begin{bmatrix} \text{yellow square} \\ \text{yellow square} \\ \text{yellow square} \end{bmatrix} \\ \text{index1} &= [1 \ 2 \ 3] & \text{index2} &= [1 \ 1 \ 1] \end{aligned}$$

- Using the index returned from max along with the its linear index in the magnitude vector yields its subscript.
- e.g., `yellow square`'s linear index in `mag1` is 1. Since we used `dim = 1`, this is its column number.
- The row number will be the index returned by max for `yellow square`, `index1(1) = 1`.
- Incidentally, `yellow square` can be found at `C(1,1)`.