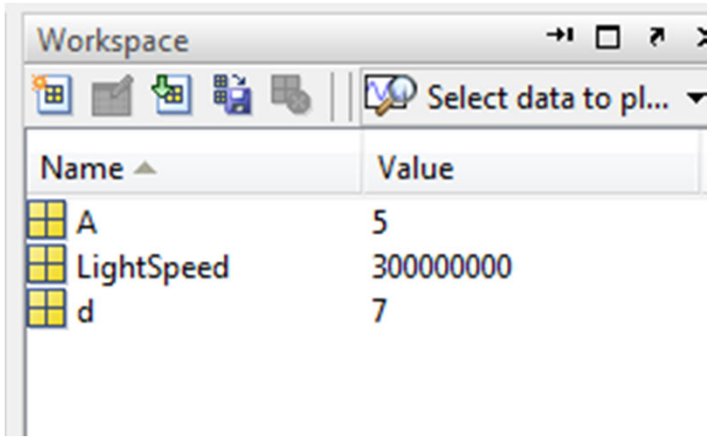
Lecture 2: Variables, Vectors and Matrices in MATLAB

Variables in MATLAB

- Just like other programming languages, you can define variables in which to store values.
- All variables can by default hold matrices with scalar or complex numbers in them.
- You can define as many variables as your PC memory can hold.
- Values in variables can be inspected, used and changed
- Variable names are case-sensitive, and show up in the Workspace.

```
>> A = 5
A =
     5

>> d = 7
d =
     7

>> LightSpeed = 3e8
LightSpeed =
   300000000
```



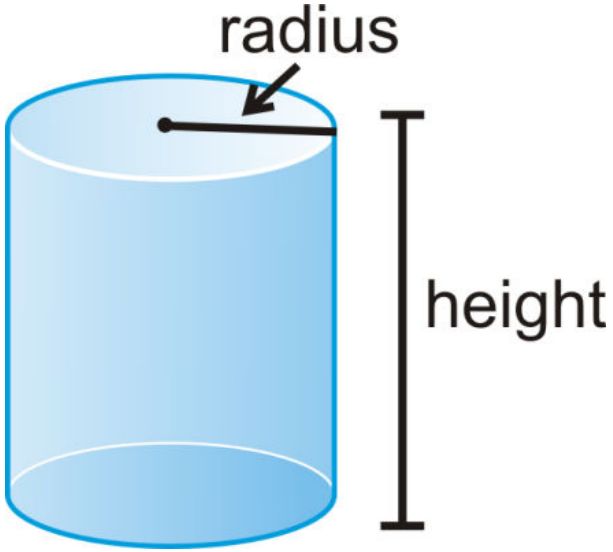
Variables

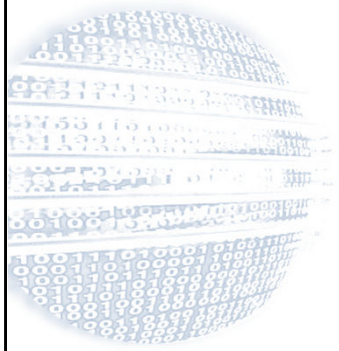
- You can change the value in the variable by over-writing it with a new value
- Remember that variables are case-sensitive (easy to make a mistake)
- Always left-to right
  >> variable = expression

```
>> a = 7
a =
     7

>> b = 12
b =
    12

>> b = 14
b =
    14

>> B = 88
B =
    88

>> c = a + b
c =
    21

>> c = a / b
c =
    0.5000
```

Exercise

- Develop MATLAB code to find Cylinder volume and surface area.
- Assume radius of 5 m and height of 13 m.





Solution

```
>> r = 5
r =
     5

>> h = 13
h =
    13

>> Volume = pi * r^2 * h
Volume =
  1.0210e+003

>> Area = 2 * pi * r * (r + h)
Area =
  565.4867
```

Useful MATLAB commands

Command	Description
`clc`	Clears the Command window.
`clear`	Removes all variables from memory.
`clear var1 var2`	Removes the variables `var1` and `var2` from memory.

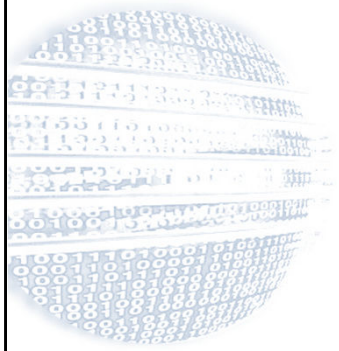
Vectors and Matrices (Arrays)

- So far we used MATLAB variables to store a single value.
- We can also create MATLAB arrays that hold multiple values
  - List of values (1D array) called **Vector**
  - Table of values (2D array) called **Matrix**
- Vectors and matrices are used extensively when solving engineering and science problems.

Row Vector

- Row vectors are special cases of matrices.
- This is a 7-element row vector (1×7 matrix).
- Defined by enclosing numbers within square brackets `[ ]` and separating them by `,` or a space.

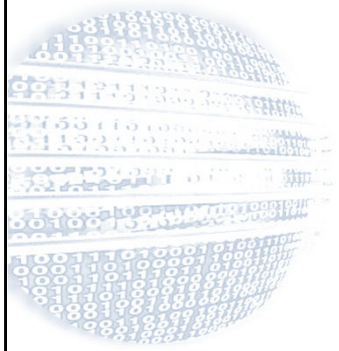
```
>> C = [10, 11, 13, 12, 19, 16, 17]

C =
    10    11    13    12    19    16    17


>> C = [10 11 13 12 19 16 17]

C =
    10    11    13    12    19    16    17
```



Column Vector

- Column vectors are special cases of matrices.
- This is a 7-element column vector (7×1 matrix).
- Defined by enclosing numbers within `[ ]` and separating them by semicolon `;`

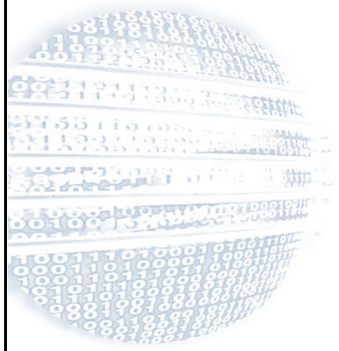
```
>> R = [10; 11; 13; 12; 19; 16; 17]

R =
     10
     11
     13
     12
     19
     16
     17
```



Matrix

- This is a 3×4-element matrix.
- It has 3 rows and 4 columns (dimension 3×4).
- Spaces or commas separate elements in different columns, whereas semicolons separate elements in different rows.
- A dimension $n \times n$ matrix is called *square* matrix.

```
>> M = [1, 3, 2, 9; 6, 7, 8, 1; 7, 4, 6, 0]
M =
     1     3     2     9
     6     7     8     1
     7     4     6     0


>> M = [1 3 2 9; 6 7 8 1; 7 4 6 0]
M =
     1     3     2     9
     6     7     8     1
     7     4     6     0
```



Transpose of a Matrix

- The transpose operation interchanges the rows and columns of a matrix.
- For an $m \times n$ matrix $\mathbf{A}$ the new matrix $\mathbf{A}^T$ (read "A transpose") is an $n \times m$ matrix.
- In MATLAB, the `A'` command is used for transpose.

$$\mathbf{A} = \begin{bmatrix} -2 & 6 \\ -3 & 5 \end{bmatrix} \qquad \mathbf{A}^T = \begin{bmatrix} -2 & -3 \\ 6 & 5 \end{bmatrix}$$

Exercise

```
>> A = [1 2 3; 5 6 7]
A =
     1     2     3
     5     6     7

>> A'
ans =
     1     5
     2     6
     3     7
```

```
>> B = [5 6 7 8]
B =
     5     6     7     8

>> B'
ans =
     5
     6
     7
     8
```

- What happens to a row vector when transposed?
- What happens to a column vector when transposed?

Useful Functions

`length(A)`	Returns either the number of elements of A if A is a vector or the largest value of m or n if A is an $m \times n$ matrix
`size(A)`	Returns a row vector `[m n]` containing the sizes of the $m \times n$ matrix A.
`max(A)`	For vectors, returns the largest element in A. For matrices, returns a row vector containing the maximum element from each column.

More Useful Functions

`sort(A)`	Sorts each column of the array `A` in ascending order and returns an array the same size as `A`.
`sum(A)`	Sums the elements in each column of the array `A` and returns a row vector containing the sums.

Exercises

```
>> X = [4 9 2 5]
X =
     4     9     2     5

>> length(X)
ans =
     4

>> size(X)
ans =
     1     4

>> min(X)
ans =
     2
```

```
>> M = [1 6 4; 3 7 2]

>> size(M)

>> length(M)

>> max(M)

>> [a,b] = max(M)

>> sort(M)



>> sum(M)
```

Solution

```
>> M = [1 6 4; 3 7 2]
M =
     1     6     4
     3     7     2

>> size(M)
ans =
     2     3

>> length(M)
ans =
     3

>> max(M)
ans =
     3     7     4
```

```
>> sort(M)
ans =
     1     6     2
     3     7     4

>> sum(M)
ans =
     4    13     6
```

Creating Big Matrices

- What if you want to create a Matrix that contains 1000 element (or more)?
- Writing each element by hand is difficult, time-consuming and error-prone.
- MATLAB allows simple ways to quickly create matrices, such as:
- Using the colon : operator (very popular).

Using the colon operator

- MATLAB command `X = J:D:K`

- In other words, it creates a vector X of values **starting** at J, **ending** with K, and with **spacing** D.
- Notice that the last element is K if K - J is an integer multiple of D. If not, the last value is *less than* J.
- MATLAB command `J:K` is the same as `J:1:K`.
- Note:
  - `J:K` is empty if J > K.
  - `J:D:K` is empty if D == 0, if D > 0 and J > K, or if D < 0 and J < K.

Example 1

```
>> x = 0:2:8
x =
     0     2     4     6     8

>> x = 0:2:7
x =
     0     2     4     6

>> x = 4:7
x =
     4     5     6     7

>> x = 7:2
x =
   Empty matrix: 1-by-0
```

Example 2

```
>> x = 7:-1:2
x =
     7     6     5     4     3     2

>> x = 5:0.1:5.9
x =
  Columns 1 through 5
    5.0000    5.1000    5.2000    5.3000    5.4000

  Columns 6 through 10
    5.5000    5.6000    5.7000    5.8000    5.9000
```

Special: ones, zeros, rand

```
>> a = ones(2,4)
a =
     1     1     1     1
     1     1     1     1

>> b = zeros(4, 3) % null matrix
b =
     0     0     0
     0     0     0
     0     0     0
     0     0     0

>> c = rand(2, 4)
c =
    0.8147    0.1270    0.6324    0.2785
    0.9058    0.9134    0.0975    0.5469

% random values drawn from the standard
% uniform distribution on the open
% interval(0,1)
```

Null and Identity Matrix

```
>> eye(4) % identity matrix
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9

>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1

>> A*I
ans =
     1     2     3
     4     5     6
     7     8     9
```

$$0A = A0 = 0$$

$$IA = AI = A$$

Matrix Determinant & Inverse

```
>> A = [1 2 3; 2 3 1; 3 2 1]
A =
     1     2     3
     2     3     1
     3     2     1

>> det(A) % determinant
ans =
   -12

>> inv(A) % inverse
ans =
   -0.0833   -0.3333    0.5833
   -0.0833    0.6667   -0.4167
    0.4167   -0.3333    0.0833

>> A^-1
ans =
   -0.0833   -0.3333    0.5833
   -0.0833    0.6667   -0.4167
    0.4167   -0.3333    0.0833
```

Accessing Matrix Elements

```
>> C = [10, 11, 13, 12, 19, 16, 17]

C =
    10    11    13    12    19    16    17

>> C(4)
ans =
    12

>> C(1,4)
ans =
    12

>> C(20)
??? Index exceeds matrix dimensions.
```

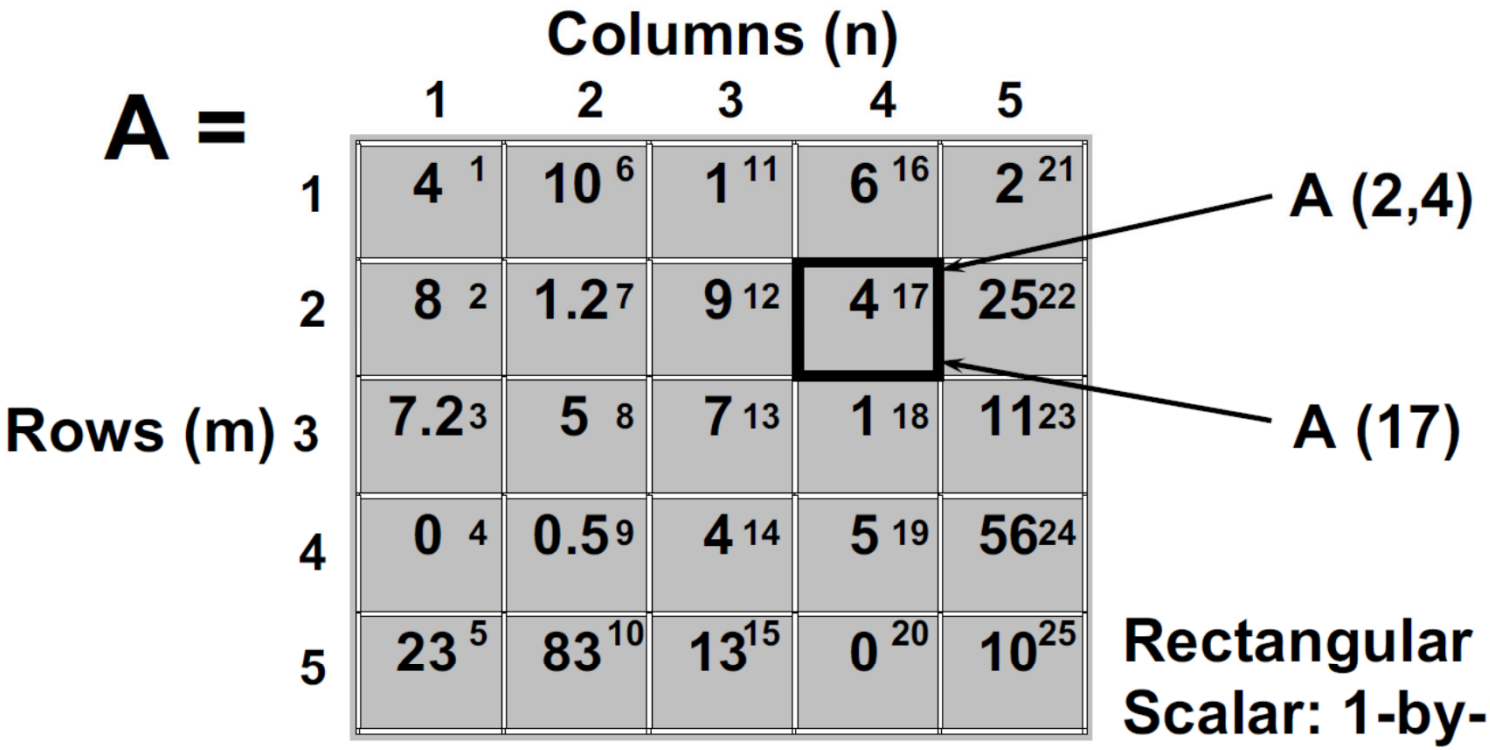
Notes

- Use `()` not `[]` to access matrix elements.
- The row and column indices are NOT zero-based, like in C/C++.
- The first is row number, followed by the column number.
- For matrices and vectors, you can use one of three indexing methods: matrix row and column indexing; linear indexing

Accessing Matrix Elements

```
>> M = [1, 3, 2, 9; 6, 7, 8, 1; 7, 4, 6, 0]
M =
     1     3     2     9
     6     7     8     1
     7     4     6     0

>> M(2, 3)
ans =
     8

>> M(3, 1)
ans =
     7

>> M(0, 1)
??? Subscript indices must either be real
positive integers or logicals.

>> M(9)
ans =
     6
```

Matrix Linear Indexing



A = 5 x 5 matrix.

Rectangular Matrix:
Scalar: 1-by-1 array
Vector: m-by-1 array
1-by-n array
Matrix: m-by-n array

Indexing: Sub-matrix

- `v(2:5)` represents the second through fifth elements
  - i.e., v(2), v(3), v(4), v(5).
- `v(2:end)` represents the second till last element of v.

- `A(:,3)` denotes all elements in the third column of matrix A.
- `A(:,2:5)` denotes all elements in the second through fifth columns of A.
- `A(2:3,1:3)` denotes all elements in the second and third rows that are also in the first through third columns.
- `A(end,:)` all elements of the last row in A.
- `A(:,end)` all elements of the last column in A.
- `v = A(:)` creates a vector v consisting of all the columns of A stacked from first to last.

Exercise

```
>> v = 10:10:70
v =
    10    20    30    40    50    60    70

>> v(2:5)
ans =
    20    30    40    50

>> v(2:end)
ans =
    20    30    40    50    60    70

>> v(:)
ans =
    10
    20
    30
    40
    50
    60
    70
```

Exercise

```
>> A = [4 10 1 6 2; 8 1.2 9 4 25; 7.2 5 7 1
11; 0 0.5 4 5 56; 23 83 13 0 10]

A =
    4.0000   10.0000    1.0000    6.0000    2.0000
    8.0000    1.2000    9.0000    4.0000   25.0000
    7.2000    5.0000    7.0000    1.0000   11.0000
         0    0.5000    4.0000    5.0000   56.0000
   23.0000   83.0000   13.0000         0    0.0000

>> A(:,3)
ans =
     1
     9
     7
     4
    13

>> A(:,2:5)
ans =
    10.0000    1.0000    6.0000    2.0000
     1.2000    9.0000    4.0000   25.0000
     5.0000    7.0000    1.0000   11.0000
     0.5000    4.0000    5.0000   56.0000
    83.0000   13.0000         0   10.0000

>> A(2:3,1:3)
ans =

     8.0000    1.2000    9.0000
     7.2000    5.0000    7.0000
```

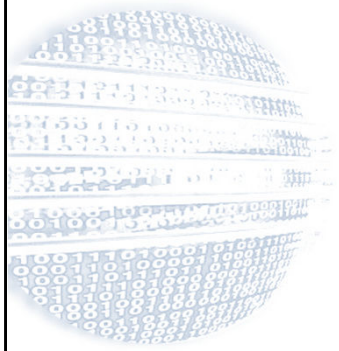
```
>> A(end,:)
ans =
    23    83    13     0    10

>> A(:,end)
ans =
     2
    25
    11
    56
    10

>> v = A(:)
v =
     4.0000
     8.0000
     7.2000
          0
    23.0000
    10.0000
     1.2000
     5.0000
     0.5000
    83.0000
     1.0000
     9.0000
     7.0000
     4.0000
    13.0000
     6.0000
     4.0000
     1.0000
     5.0000
          0
     2.0000
    25.0000
    11.0000
    56.0000
    10.0000
```

Linear indexing is useful: `find`

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9

>> B = find(A > 5) % returns linear index
B =
     3
     6
     8
     9
```

Extending Matrices

- You can add extra elements to a matrix by creating them directly using `( )`
- Or by concatenating (appending) them using `[ , ]` or `[ ; ]`
- If you don't assign array elements, MATLAB gives them a default value of 0



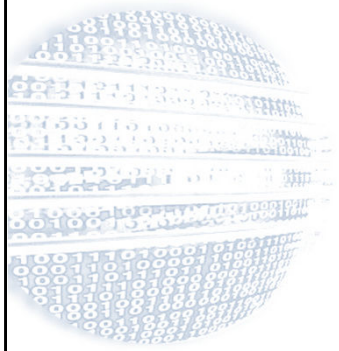
```
>> h = [12 11 14 19 18 17]
h =
   12   11   14   19   18   17

>> h = [h 13]
h =
   12   11   14   19   18   17   13

>> h(10) = 1
h =
   12   11   14   19   18   17   13    0    0    1
```

Example

```
>> a = [2 4 20]
a =
     2     4    20

>> b = [9, -3, 6]
b =
     9    -3     6

>> [a b]
ans =
     2     4    20     9    -3     6

>> [a, b]
ans =
     2     4    20     9    -3     6

>> [a; b]
ans =
     2     4    20
     9    -3     6
```

Functions on Arrays

- Standard MATLAB functions (sin, cos, exp, log, etc) can apply to vectors and matrices as well as scalars.



```
>> x = [1, 2, 3]
x =
     1     2     3

>> y = sin(x)
y =
    0.8415    0.9093    0.1411
```