Chapter 3. Arrays and Array Operations
Outline

- 3.1 Introduction
- 3.2 Types of Arrays and Indexing Their Elements
- 3.3 Array Operations
- 3.4 Array Functions
- 3.5 Using MATLAB Matrix Formalism to Handle Polynomials
- 3.6 Handling Character and String Arrays
The Empty Matrix (0-by-0 matrix)

Scalar (1-by-1 matrix)

Vector (1-by-n or n-by-1 matrix)

Raw Vector (1-by-n matrix)  Column Vector (n-by-1 matrix)

Zero Vector  Unit Vector  Zero Vector  Unit Vector

Matrix (n-by-m matrix)

Rectangular Matrix (n-by-m matrix)  Square Matrix (n-by-n matrix)

Diagonal Matrix  Triangular Matrix  Magic Matrix  Orthogonal Matrix

Identity Matrix  Block Diagonal Matrix  Orthonormal Matrix

a = []
Matrices

\[
\begin{bmatrix}
11 & 12 & 13 & 14 & 15 & 16 \\
21 & 22 & 23 & 24 & 25 & 26 \\
31 & 32 & 33 & 34 & 35 & 36 \\
41 & 42 & 43 & 44 & 45 & 46
\end{bmatrix}
\]

\[
a = [a_{ij}]
\]

\[
a = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46}
\end{bmatrix}
\]
Examples of 1-, 2- and 3-D Arrays

1D arrays

Row 1
Row 2
Row 3
Row 4

Col 1 Col 2 Col 3 Col 4 Col 5 Col 6

a_{1 \times 6}

2D array

Row 1
Row 2
Row 3
Row 4

Col 1 Col 2 Col 3 Col 4 Col 5 Col 6

a_{4 \times 1}

a_{4 \times 6}

3D array

Col 1 Col 2 Col 3 Col 4 Col 5 Col 6

Row 1 Row 2 Row 3 Row 4

Col 6

Page 1 Page 2 Page 3 Page 4 Page 5

a_{4 \times 6 \times 5}
Creating Arrays: Quick Start

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

a =

    1    2    3    4

>> b=a'

b =

    1
    2
    3
    4

>> c=[7; 8; 6; 9]

c =

    7
    8
    6
    9

>> f=[1,2,3,5; 4 5 8 9]

f =

    1    2    3    5
    4    5    8    9
```

```
>> g=linspace(1,10,3)

g =

    1.0000    5.5000   10.0000

>> k1=linspace(1,4,4):
   logspace(1,3,4)]

k1 =

    1.0e+003 *

     0.0010    0.0020    0.0030    0.0040
     0.0100    0.0464    0.2154    1.0000

>> k1(:,:,2)=rand(2,4)

k1(:,:,1) =

    1.0e+003 *

     0.0010    0.0020    0.0030    0.0040
     0.0100    0.0464    0.2154    1.0000

k1(:,:,2) =

    0.9575    0.1576    0.9572    0.8003
    0.9649    0.9706    0.4854    0.1419
```
Array Dimensions
Addressing a Single Element

1D arrays

Row 1
Row 2
Row 3
Row 4

Col 1 Col 2 Col 3 Col 4 Col 5 Col 6

a(1, 5)

a(4, 1)

2D array

Row 1
Row 2
Row 3
Row 4

Col 1 Col 2 Col 3 Col 4 Col 5 Col 6

a(2, 4)

3D array

Row 1
Row 2
Row 3
Row 4

Col 1 Col 2 Col 3 Col 4 Col 5 Col 6

Page 1
Page 2
Page 3
Page 4
Page 5

a(1, 4, 4)
Arrays in Computer Memory

\[ a(3,2) \]

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\end{array}
\]

\[ i = \text{sub2ind}(\text{size}(a), 3, 2) \] returns \( i = 6 \)

\[ [r, c] = \text{ind2sub}(\text{size}(a), 6) \] returns \( r = 3 \) and \( c = 2 \)

\[ \text{size}(a) \] returns \([3, 4]\)

\[ \text{length}(a) \] returns 4 (it is equivalent to \( \max(\text{size}(X)) \))

\[ \text{ndims}(a) \] returns 2

\[ \text{numel}(a) \] returns 12

\[
\begin{array}{cccc}
1 & A(1,1) \\
5 & A(2,1) \\
9 & A(3,1) \\
2 & A(1,2) \\
6 & A(2,2) \\
10 & A(3,2) \\
3 & A(1,3) \\
7 & A(2,3) \\
11 & A(3,3) \\
4 & A(1,4) \\
8 & A(2,4) \\
12 & A(3,4) \\
\end{array}
\]
Addressing Multiple Elements

sub2ind(size(a), [4 2], [2 3])
Creating Special Arrays

1) `ones(m,n,p,...)` creates an $m \times n \times p \times \ldots$ array of ones (`ones(n)` creates an $n \times n$ matrix of ones)

2) `zeros(m,n,p,...)` creates an $m \times n \times p \times \ldots$ array of zeros (`zeros(n)` creates an $n \times n$ matrix of zeros, called a null matrix and denoted by 0)

3) `eye(m,n)` creates an $m \times n$ matrix with 1’s on the diagonal and zeros elsewhere (`eye(n)` creates a square $n \times n$ matrix, called identity matrix and denoted by I, where all the elements are zero except the diagonal elements, which are unity)

4) `magic(n)` creates a magic square constructed from the integers 1 through $n^2$ with equal row, column, and diagonal sums (produces valid magic squares for all $n > 0$ except $n = 2$)

5) `pascal(n)` creates the Pascal matrix of order $n$ (a symmetric positive definite matrix with integer entries), where any element is computed from Pascal’s triangle, i.e. $a_{i,j} = a_{i-1,j} + a_{i,j-1}$

```
>> ones(1,4)
ans =
     1     1     1     1

>> magic(6)
ans =
     35     1     6    26    19    24
      3    32     7    21    23    25
     31     9     2    22    27    20
     8    28    33    17    10     1
    30     5    34    12    14    16
    4     36    29     13    18    11
```

See `gallery` for creating more test matrices
Assessing Array Properties

1) \texttt{ndims(A)} returns the number of dimensions in the array \texttt{A}

2) \texttt{size(A)} returns a row vector whose elements contain the length of the corresponding dimension of \texttt{A} (\texttt{size(A,dim)} returns the length of dimension \texttt{dim}; \texttt{[m,n] = size(A),…}) returns the number of rows and columns when \texttt{A} is a matrix)

3) \texttt{length(A)} returns the length of the largest array dimension in \texttt{A} (for vectors the length is simply the number of elements, otherwise it is equivalent to \texttt{max(size(X))})

4) \texttt{numel(A)} returns the number of elements in array \texttt{A} (equivalent to \texttt{prod(size(A))})

\begin{verbatim}
>> A=pascal(3)
A =
  1  1  1
  1  2  3
  1  3  6

>> A(:,1)=[]
A =
  1  1
  2  3
  3  6

>> ndims(A) >> size(A)
ans =
  2
ans =
  3  2

>> length(A) >> numel(A)
ans =
  3
ans =
  6
\end{verbatim}
Arrays of Random Numbers

1) \texttt{rand(m,n,p,...)} creates \( mxnxp \times \ldots \) array with uniformly distributed (on the interval (0.0,1.0)) random entries
   - \texttt{rand(n)} creates an \( nxn \) matrix with random entries

2) \texttt{randn(m,n,p,...)} creates \( mxnxp \times \ldots \) array with normally distributed (chosen from a normal distribution with mean zero, variance one and standard deviation one) random entries (try \texttt{hist(randn(10000,1),20)})
   - \texttt{randn(n)} creates an \( nxn \) matrix with random entries

3) \texttt{randi(imax,m,n,p,...)} creates \( mxnxp \times \ldots \) array containing pseudorandom integer values drawn from the discrete uniform distribution on \( 1:imax \)
   - \texttt{randi(imax)} generates a single number
   - \texttt{randi(imax,n)} creates a square \( nxn \) matrix
   - \texttt{randi([imin,imax],...)} returns an array containing integers drawn from the discrete uniform distribution on the interval \([imin,imax]\)

\[
\begin{align*}
\text{>> rand(2,3,2)} & \quad \text{>> randi(5,4)} \\
\text{ans(:,1)} & = \\
0.8147 & 0.1270 & 0.6324 \\
0.9058 & 0.9134 & 0.0975 \\
\text{ans(:,2)} & = \\
0.2785 & 0.9575 & 0.1576 \\
0.5469 & 0.9649 & 0.9706 \\
\text{ans} & = \\
5 & 3 & 4 & 4 \\
3 & 5 & 1 & 4 \\
5 & 4 & 5 & 4 \\
1 & 5 & 5 & 2
\end{align*}
\]
Open a new MATLAB session and type

```matlab
>> rand(1,3)
```

to get

```
ans =
  0.8147  0.9058  0.1270
```

Now, close/reopen MATLAB and try

```matlab
>> rng('shuffle'), rand(1,3)
```

The `rng` function allows controlling random number generation. Some common syntax are

```
rng(seed)
rng('shuffle')
rng('default')
rng(s)
```

% Get generator settings
s = rng;
% Call rand
x = rand
% Restore previous generator settings
rng(s);
% Call rand again and get the same results
y = rand

```
x =  
0.1576

y =  
0.1576
```
1) \texttt{diag(v)} puts elements of vector \texttt{v} on the main diagonal of a square matrix
2) \texttt{diag(v,k)} puts \( n \) elements of vector \texttt{v} on the \( k^{th} \) diagonal of a square matrix of order \( n+\text{ABS}(k) \) (for the main diagonal \( k=0 \))
3) \texttt{diag(X)} returns a column vector formed from elements of the main diagonal of matrix \texttt{X}
4) \texttt{diag(X,k)} returns a column vector formed from the elements of the \( k^{th} \) diagonal of matrix \texttt{X}
5) \texttt{blkdiag(A,B,C,...)}, where \texttt{A}, \texttt{B}, \texttt{C}, ... are matrices, outputs a block diagonal matrix
Concatenation Functions

1) `cat(1,A,B)` or `vertcat(A,B)` is equivalent to `[A;B]`
2) `cat(2,A,B)` or `horzcat(A,B)` is equivalent to `[A,B]`

Matrix A

Matrix B

C = `cat(1,A,B)`

C = `cat(2,A,B)`

C = `cat(3,A,B)`
Creating Triangular Matrices

1) \( \text{triu}(B,k) \) returns the upper triangular part of matrix \( B \) (on and above the \( k \)th diagonal of \( B \))

2) \( \text{tril}(B,k) \) returns the lower triangular part of matrix \( B \) (on and below the \( k \)th diagonal of \( B \))
\texttt{repmat(A,m,n)} creates a matrix consisting of an \textit{m}-by-\textit{n} tiling of copies of \texttt{A}. The size of this matrix is [\texttt{size(A,1)*m, size(A,2)*n}] (the statement \texttt{repmat(A,n)} creates an \textit{n}-by-\textit{n} tiling).
Ts=0.1;
triangle = repmat([0:Ts:(1-Ts),1:-Ts:(0+Ts)],1,4);
subplot(2,1,1)
plot(triangle)

pulsetrain = repmat([ones(1,125),zeros(1,125)],1,4);
time = 0:Ts:Ts*(length(pulsetrain)-1);
subplot(2,1,2)
stairs(time,pulsetrain)

Of course, Signal Processing Toolbox features a variety of built-in MATLAB functions creating different signals:
- square
- sawtooth
- gauspuls
- rectpuls
- sinc
- diric
- tripuls
Reshaping

1) \texttt{reshape}(A,m,n) \text{ returns the } m\text{-by-}n \text{ matrix, whose elements are taken columnwise from } A \text{ (an error results if } A \text{ does not have } m \times n \text{ elements)}

2) \texttt{reshape}(A,m,n,p,...) \text{ returns the } n\text{D array, whose elements are taken columnwise from } A \text{ (} m \times n \times p \times ... \text{ must be the same as } \text{prod(size(A)) meaning there is a correct number of elements to do this)}

One more option for reshaping the existing arrays is to use the empty matrix. For instance, having matrix \( A \) as

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{bmatrix}
\]

and issuing the following two commands:

\[
\text{A( : , [2,4] ) = [ ]; A(3,:)=[ ]}
\]

returns

\[
\begin{bmatrix}
1 & 6 & 11 \\
5 & 10 & 4 \\
9 & 3 & 8 \\
2 & 7 & 12
\end{bmatrix}
\]

\[
\begin{bmatrix}
3 & 11 & 8 \\
7 & 4 & 12 \\
1 & 9 & 6 \\
5 & 2 & 10
\end{bmatrix}
\]

\[
\begin{bmatrix}
12 \\
11 \\
9 \\
10 \\
8 \\
7 \\
5 \\
6 \\
4 \\
3 \\
1 \\
2
\end{bmatrix}
\]
Permutations

1) **randperm(n,k)** returns a row vector containing **k** unique integers selected randomly from 1 to **n** inclusive

2) **perms(v)** returns a matrix containing all permutations of the elements of vector **v** in reverse lexicographic order

3) **circshift(A,k)** circularly shifts the elements in array **A** by **k** positions

```
>> randperm(5)
an =
    2     5     4     3     1
>> randperm(10,3)
an =
    9     5     4

>> perms([1 3 5])
an =
    5     3     1
    5     1     3
    3     5     1
    3     1     5
    1     5     3
    1     3     5

>> circshift([1 2 3 4],1)
an =
    4     3     1     2
```

Also, see **permute**
• double-click a variable in the Workspace browser, or
• use `openvar variablename` command

Selected elements

Applicable functions to visualize the selected cells

View and change values of the array elements

As of R2012b
Visualization of Data Points

```matlab
R = randi([2,15],10,1);
plot(R)
```

```matlab
stem(R)
plot(R,'o')
scatter(1:length(R),R)
```
z = ones(100, 200);
z(10:50, 150:170) = 0;
z(70:80, 20:35) = 0;
z(4:5:100, 50:130) = 0;
z(3:5:100, 50:130) = 0;
z(2:5:100, 50:130) = 0;
z = z - eye(100, 200);
spy(z)

imagesc(z)
colorbar
3D Visualization of a Matrix

\[
w = 0.5 \times \text{eye}(20);
\]
\[
w(5:15, 5:15) = -0.2 \times \text{ones}(11);
\]
\[
\text{meshc}(w)
\]

\[
[x, y] = \text{meshgrid}(-3:0.125:3);
\]
\[
z = \text{peaks}(x, y);
\]
\[
z(15:35, 15:35) = \text{NaN};
\]
\[
\text{meshc}(x, y, z)
\]
Visualization of Random Data Sample

```matlab
histogram(rand(100,1))
```

```matlab
hist(rand(1000,10),20)
```
## Scalar, Matrix and Cross Product Two Matrices

### Matrix product

\[
F = A_{3\times1} B_{1\times3} = [f_{ij}] = AB = \begin{bmatrix}
a_{1b_1} & a_{1b_2} & a_{1b_3} \\
a_{2b_1} & a_{2b_2} & a_{2b_3} \\
a_{3b_1} & a_{3b_2} & a_{3b_3}
\end{bmatrix}
\]

\[
\begin{align*}
\text{>> } & \quad F = [1 \ 2 \ 3]' \ast [4 \ 3 \ 2] \\
F &= \\
&4 \ 3 \ 2 \\
&8 \ 6 \ 4 \\
&12 \ 9 \ 6
\end{align*}
\]

### Cross product

\[
K = A \times B = [a_2b_3 - a_3b_2 \quad a_3b_1 - a_1b_3 \quad a_1b_2 - a_2b_1]
\]

### Scalar product

\[
E = A_{1\times n}B_{n\times1} = [e_{ij}] = AB = \sum_{k=1}^{n} a_{ik}b_{kj}
\]

\[
\begin{align*}
\text{>> } & \quad E = \text{dot}([1 \ 2 \ 3 \ 4], [4 \ 3 \ 2 \ 1]) \\
E &= 20 \\
\text{>> } & \quad E = [1 \ 2 \ 3 \ 4]' \ast [4 \ 3 \ 2 \ 1]' \\
E &= 20
\end{align*}
\]

\[
A = [1.5, 0, 0]; \quad B = [0, 0.8, 0];
\]

\[
K = \text{cross}(A, B);
\]

\[
\begin{align*}
\text{quiver3}(0, 0, 0, A(1), A(2), A(3), 'b', 'Linewidth', 3) \\
\text{hold} \\
\text{quiver3}(0, 0, 0, B(1), B(2), B(3), 'g--', 'Linewidth', 3) \\
\text{quiver3}(0, 0, 0, K(1), K(2), K(3), 'r--', 'Linewidth', 3) \\
\text{axis equal} \\
\text{xlabel('axis x')} \\
\text{ylabel('axis y')} \\
\text{zlabel('axis z')} \\
\text{view([-140 20])} \\
\text{legend('Vector A', 'Vector B', 'cross(A,B)', 'location', 'nw')}
\end{align*}
\]
## Non-Standard (element-wise) Array Operations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>MATLAB Form</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exponentiation, $a^b$</td>
<td>$a.^b$</td>
</tr>
<tr>
<td>*</td>
<td>multiplication, $ab$</td>
<td>$a.*b$</td>
</tr>
<tr>
<td>/</td>
<td>right division, $a/b$</td>
<td>$a./b$</td>
</tr>
<tr>
<td>\</td>
<td>left division, $a\b$</td>
<td>$a\b$</td>
</tr>
</tbody>
</table>

### Example

```matlab
>>> a = 2*ones(2,3)
a =
    2   2   2
    2   2   2
>>> b = rand(2,3)
b =
    0.7922   0.6557   0.8491
    0.9595   0.0357   0.9340
>>> a.*b
Error using *
Incorrect dimensions for matrix multiplication.
Check that the number of columns in the first matrix matches the number of rows in the second matrix. To perform elementwise multiplication, use '.*'.

>>> a./b
ans =
    2.5246   3.0500   2.3554
    2.0844  56.0041  2.1413
```

```matlab
>>> a.^b
ans =
    1.5844   1.3115   1.6983
    1.9190   0.0714   1.8680
>>> a\b
ans =
    1.7317   1.5754   1.8014
    1.9446   1.0251   1.9106
```
The Beauty of MATLAB

clear all, clc
a=rand(1000,10000); b=rand(1000,10000);
tic
c=a.^b;
toc

Elapsed time is 4.711545 seconds.

clear c
tic
for i=1:1000
    for j=1:10000
        c(i,j)=a(i,j)^b(i,j);
    end
end
toc

Elapsed time is 113.778294 seconds.

clear c
tic
c=NaN(1000,10000);
for i=1:1000
    for j=1:10000
        c(i,j)=a(i,j)^b(i,j);
    end
end
toc

Elapsed time is 8.185135 seconds.
1) sort(A) sorts the elements of A along the first non-singleton dimension of an array; sort(A,dim) sorts along the dimension dim; sort(..., 'descend') sorts the elements in the descending order

2) sortrows(A) sorts the rows of A in ascending order of the first column; sortrows(A,column) sorts matrix A based on the columns specified in the vector column

3) flip(A) returns an array of the same size as A, but with the order of elements reversed (flip(A,dim) reverses the order of elements in A along the dimension dim)
   - flipud(A) returns A with its rows flipped in the up-down direction
   - fliplr(A) returns A with its columns flipped in the left-right direction

4) rot90(A,k) rotates A counterclockwise by k*90 degrees, where k is a signed integer

>> a=magic(4); a(:,2)=[]
a =
   16     3    13
    5    10     8
     9     6    12
     4    15     1
>> sort(a)
ans =
   4     3     1
    5     6     8
     9    10    12
    16    15    13
>> sort(a,2)
ans =
     3    13    16
     5     8    10
     6     9    12
     1     4    15
>> sortrows(a)
ans =
   4    15     1
    9     6    12
    5    10     8
   16     3    13
>> flip(a)
ans =
   4    15     1
    9     6    12
    5    10     8
   16     3    13
>> fliplr(a)
ans =
   13     3    16
     8    10     5
    12     6     9
     8    10     5
>> rot90(a,-90)
ans =
    1     5    15
     8    10     5
    12     6     9
    13     3    16
sum, min and max Functions

1) \texttt{sum(A)} sums the elements of each column of \textit{A} and returns a row vector with these sums; if \textit{A} is a multidimensional array, \texttt{sum(A)} treats the values along the first non-singleton dimension as vectors, returning an array of row vectors; \texttt{sum(A,dim)} sums along a specified dimension \textit{dim} (\texttt{'all'}, scalar or vector); \texttt{sum(\ldots,'omitnan')} specifies to omit NaN values from calculations

2) \texttt{cumsum(A)} returns the cumulative sum of \textit{A} (other syntaxes are similar to \texttt{sum})

3) \texttt{min(A,[ ],dim)} returns the smallest value along a specified dimension \textit{dim} (\texttt{'all'}, scalar or vector); \texttt{min(\ldots, 'includenan')} specifies to include NaN values (by default they are excluded, which is opposite to \texttt{sum})

2) \texttt{[x,k]=min(\ldots)} returns minimum values in row vector \textit{x} along with indices in a row vector \textit{k}

3) \texttt{min(A,B)} returns an array the same size as \textit{A} and \textit{B} with the smallest elements taken from \textit{A} or \textit{B} (the dimensions of \textit{A} and \textit{B} must match, or they may be scalar)

4) \texttt{max(\ldots)} is similar to \texttt{min(\ldots)}, but using maximum values

\begin{verbatim}
>> sum(a)   % 1) sum of all elements in a
ans =
       34
       34
       34

>> min(a,[],2)  % 2) min of each column of a
ans =
       3
       5

>> [res,ind]=max(a)  % 3) max, res = max values, ind = indices
res =
        16
        15
        13
ind =
        1
        4
        1

>> cumsum([1 2 3])  % 2) cumsum of [1 2 3]
ans =
        1
        3
        6

>> [v,i]=min(a(:))  % 2) min of all elements in a(:)
ans =
        1
        1

>> max(a,[],'all')  % 4) max of all elements in a, 'all'
ans =
        16

Also, see \texttt{prod}, \texttt{cumprod}
\end{verbatim}
Statistical Functions

1) \([s,l]=\text{bounds}(...)\) returns the smallest element \(s\) (equivalent to \(\text{min}(...)\)) and largest element \(l\) (equivalent to \(\text{max}(...)\)) of an array with all syntaxes similar to that of \(\text{min}\).

2) \(\text{mean}(A,\text{dim})\) returns the mean values of the elements \((\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i)\) along a specified dimension \(\text{dim}\) (‘all’, scalar or vector); \(\text{mean}(..., \text{‘omitnan’})\) specifies to omit NaN values from calculations.

2) \(\text{median}(...)\) returns the median values of the elements along different dimensions of an array with the syntax similar to that of \(\text{mean}\).

3) \(\text{std}(A,[\ ],\text{dim})\) returns the standard deviation of the elements of \(A\) along the optional dimension \(\text{dim}\), specified as ‘all’, scalar or vector, similar to that of \(\text{mean}\). The place holder \([\ ]\) may be either 0 (default value) or 1, corresponding to square root of an unbiased estimate of the variance and the second moment of the set of values about their mean, respectively

\[
s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}
\]

\(\text{std}(..., \text{‘omitnan’})\) specifies to omit NaN values from calculations.

\[\text{>> [s,l]=bounds(a,[1 2])} \quad \text{>> median(a)} \quad \text{>> std(a,[],'all')}\]

\[
s = 1 \quad \text{res = 7 8 10} \quad \text{ans = 4.8524}
\]

\(\text{>> median(a)}\)

\[
\text{>> std(a,[],’all’)}\]

\[
s = 1 \quad \text{res = 7 8 10} \quad \text{ans = 4.8524}
\]

Also, see \text{var}, \text{mode}
The *find* Function

1) \( \text{find}(A) \) locates all nonzero elements of array \( A \), and returns their linear indices in a column vector (if \( A \) contains no nonzero elements or is an empty array, then \( \text{find}(A) \) returns an empty array)

2) \( \text{find}(A,k,\text{ind}) \) returns at most the first (if \( \text{ind} = \text{'}first\text{'} \) or omitted) or the last (\( \text{ind} = \text{'}last\text{'} \)) \( k \) indices corresponding to the nonzero entries of \( A \)

3) \([r,c,v]=\text{find}(\ldots)\) returns a vector \( v \) of the nonzero entries in matrix \( A \), as well as row \( r \) and column \( c \) indices of these elements (to see the results in the most readable way, we recommend issuing one of the following two instructions: \([r \ c \ v]\) or \(r',c',v'\))

```matlab
g >> a=magic(4); a(:,2)=[]

 >> [r,c,v]=find(a>10);

 >> disp('    row column element')
 >> disp([r,c,a(sub2ind(size(a),r,c))])
```

```
row column element
1     1    16
4     2    15
1     3    13
3     3    12
```

Also, see \texttt{nnz}, \texttt{nonzeros}
Matrix functions

1) \texttt{transpose(A)} or \texttt{.'} performs the non-conjugate transpose of matrix \texttt{A}

2) \texttt{ctranspose(A)} or \texttt{'} computes the complex-conjugate transpose of matrix \texttt{A}

\begin{verbatim}
>> a=magic(3); a(:,2)=[]     >> a.'        >> (a+i).'  
a = 8 6  
3 7  
4 2  
ans = 8 3 4  
6 7 2  
ans = 8+1i 3+1i 4+1i  
6+1i 7+1i 2+1i
\end{verbatim}

1) \texttt{det(A)} determines the determinant of a square matrix \texttt{A}

2) \texttt{inv(A)} computes the inverse of the square matrix \texttt{A} (a warning message is printed if \texttt{A} is badly scaled or nearly singular)

3) \texttt{eig(A)} returns a column vector containing the eigenvalues of square matrix \texttt{A}

4) \texttt{trace(A)} sums the diagonal elements of the square matrix \texttt{A}, which is also the sum of the eigenvalues of \texttt{A}
More Array Functions
(Unions, Intersection, Membership)

19) issorted(A) determines whether elements in A are in sorted order (see the sort function)
20) intersect(A,B) returns the data common to both A and B with no repetitions
21) ismember(A,B) returns an array containing 1 (true) where the data in A is found in B (elsewhere, it returns 0 (false) (ismembertol(A,B,tol) allows setting membership within tolerance tol)
22) setdiff(A,B) returns the data in A that is not in B
23) setxor(A,B) sets exclusive OR of two arrays, A and B
24) union(A,B) returns the combined data from A and B with no repetitions
25) unique(A) finds the unique values in the array A (uniquetol(A,tol) allows finding the unique values with tolerance tol)

```matlab
>> a=intersect([1 3 5 3 4],[3 6 1 7]) a = 1  3
>> a=ismember([1 3 5 3 4],[3 6 1 7]) a = 1  1  0  1  0
>> a=issorted([1 3 5 3 4]) a = 0
>> a=setdiff([1 3 5 3 4],[3 6 1 7]) a = 4  5
>> a=setxor([1 3 5 3 4],[3 6 1 7]) a = 4  5  6  7
>> a=union([1 3 5 3 4],[3 6 1 7]) a = 1  3  4  5  6  7
>> a=unique([1 3 5 3 4]) a = 1  3  4  5
```
Handling Polynomials

Define two polynomials, a and b, by their roots

\[ a = (x - 1)(x - 2)(x - 3) \]
\[ b = x(x - 5)(x - 6)(x - 10) \]

Multiply two polynomials \( c = a \times b \)

\[ c = \text{conv}(a, b); \]

Assure that a new polynomial has the same roots

\[ \text{polyval}(c, [r1 \ r2]) \]

Create a new polynomial \( d \)

\[ d = x(x - 5)(x - 10) \]

Divide polynomial \( c \) by polynomial \( d \)

\[ [q, r] = \text{deconv}(c, d) \]

Find the roots of the quotient polynomial

\[ \text{roots}(q) \]

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv</td>
<td>computes a product of two polynomials</td>
</tr>
<tr>
<td>deconv</td>
<td>performs a division of two polynomials (returns the quotient and remainder)</td>
</tr>
<tr>
<td>poly</td>
<td>creates a polynomial with specified roots</td>
</tr>
<tr>
<td>polyder</td>
<td>calculates the derivative of polynomials analytically</td>
</tr>
<tr>
<td>polyeig</td>
<td>solves polynomial eigenvalue problem</td>
</tr>
<tr>
<td>polyfit</td>
<td>produces polynomial curve fitting</td>
</tr>
<tr>
<td>polyint</td>
<td>integrates polynomial analytically</td>
</tr>
<tr>
<td>polyval</td>
<td>evaluates polynomial at certain points</td>
</tr>
<tr>
<td>polyvalm</td>
<td>evaluates a polynomial in a matrix sense</td>
</tr>
<tr>
<td>residue</td>
<td>converts between partial fraction expansion and polynomial coefficients</td>
</tr>
<tr>
<td>roots</td>
<td>finds polynomial roots</td>
</tr>
<tr>
<td>poly2sym</td>
<td>converts a vector of polynomial coefficients to a symbolic polynomial</td>
</tr>
<tr>
<td>sym2poly</td>
<td>converts polynomial coefficients to a vector of polynomial coefficients</td>
</tr>
</tbody>
</table>

\[ \text{pretty(poly2sym([1 2 3 4]))} \]

\[ x^3 + 2x^2 + 3x + 4 \]
pet1='cat';
number1='123';
number2=num2str(123);
number3=str2num(number1);
m=str2num(['123' '+' '17']);
l=str2num(pet1);
pet2='dog';
a=pet2(2);
b=strfind(pet1,'a');
pets1=[pet1 pet2];
pets2=[pet1; pet2];
pets3=[pet2; 'crocodile']

Error using vertcat
Dimensions of arrays being concatenated are not consistent.
pets4 = char('pet2', 'crocodile')
a1 = pets4(1,6)
a1 = pets4(2,2)
[pets4(1,1) pets4(2,3) pets4(2,1)]
ascii = char(reshape(32:127, 48, 2))

pets4 =
  2×9 char array
'pet2     '
'crocodile'
a1 =
  '
'a1 =
  'r'
an =
  'poc'
ascii =
  2×48 char array
' !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~'

The integers from 32 to 127 correspond to printable ASCII characters. However, the integers from 0 to 65535 also correspond to Unicode characters. Try

```matlab
>> C = char(8451)
C =
  '℃'
```

**Significant Whitespace Character**

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char(133)</td>
<td>Next line</td>
</tr>
<tr>
<td>char(160)</td>
<td>Nonbreaking space</td>
</tr>
<tr>
<td>char(8199)</td>
<td>Figure space</td>
</tr>
<tr>
<td>char(8239)</td>
<td>Narrow no-break space</td>
</tr>
</tbody>
</table>
Text can be represented using *string arrays* instead of *character arrays* as well. You can create a string by enclosing a piece of text in double quotes, like "text". You can also use the `string` function like `string('text')` or `string([1 2 3])`. Each element of a string array stores a sequence of characters. The sequences can have different lengths without padding. A string array that has only one element is also called a *string scalar*. You can index into, reshape, and concatenate string arrays using standard array operations, and you can append text to strings using the + operator.

```matlab
>> pet3=['dog'; 'crocodile']
pet3 = 2×1 string array    "dog"    "crocodile"
>> size(pet3)
ans = 2 1
>> pet4=char(pet3)
pet4 = 2×9 char array    'dog'    'crocodile'
>> txt=strjust(pet4,'center')
txt = 2×9 char array    '   dog   '    'crocodile'
>> pet3+pet3
ans = 2×1 string array    "dogdog"    "crocodilecrocodile"
>> txt(:,4:6)
ans = 2×3 char array    'dog'    'cod'
>> rot90(ans,-2)
ans = 2×3 char array    'doc'    'god'
```
## Character and String Array Functions

### Create and Concatenate

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char</code></td>
<td>creates a character array</td>
</tr>
<tr>
<td><code>cellstr</code></td>
<td>converts to cell array of character vectors</td>
</tr>
<tr>
<td><code>blanks</code></td>
<td>creates a character array of blanks</td>
</tr>
<tr>
<td><code>newline</code></td>
<td>creates a newline character</td>
</tr>
<tr>
<td><code>string</code></td>
<td>creates a string array</td>
</tr>
<tr>
<td><code>strings</code></td>
<td>creates array of strings with no characters</td>
</tr>
<tr>
<td><code>join</code></td>
<td>combines strings</td>
</tr>
<tr>
<td><code>compose</code></td>
<td>converts data into formatted string array</td>
</tr>
<tr>
<td><code>sprintf</code></td>
<td>formats data into string</td>
</tr>
<tr>
<td><code>append</code></td>
<td>concatenates character or string arrays horizontally</td>
</tr>
</tbody>
</table>

### Determine Type and Properties

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ischar</code></td>
<td>determines if input is character array</td>
</tr>
<tr>
<td><code>iscellstr</code></td>
<td>determines if input is cell array of character vectors</td>
</tr>
<tr>
<td><code>isstring</code></td>
<td>determines if input is string array</td>
</tr>
<tr>
<td><code>isStringScalar</code></td>
<td>determines if input is string array with one element</td>
</tr>
<tr>
<td><code>strlength</code></td>
<td>finds a length of strings in string array</td>
</tr>
<tr>
<td><code>isstrprop</code></td>
<td>determines if string is of specified category</td>
</tr>
<tr>
<td><code>isletter</code></td>
<td>determines which characters are letters</td>
</tr>
<tr>
<td><code>isspace</code></td>
<td>determines which characters are space characters</td>
</tr>
</tbody>
</table>

### Convert Input Arguments

| convertCharsToStrings | convert character arrays to string arrays, leaving other arrays unaltered |
| convertStringsToChars  | convert string arrays to character arrays, leaving other arrays unaltered |
### Character and String Array Functions

#### Edit

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase</td>
<td>deletes substrings within strings</td>
</tr>
<tr>
<td>eraseBetween</td>
<td>deletes substrings between indicators that mark starts and ends of substrings</td>
</tr>
<tr>
<td>extractAfter</td>
<td>extracts substring after specified position</td>
</tr>
<tr>
<td>extractBefore</td>
<td>extracts substring before specified position</td>
</tr>
<tr>
<td>extractBetween</td>
<td>extracts substrings between indicators that mark starts and ends of substrings</td>
</tr>
<tr>
<td>insertAfter</td>
<td>inserts string after specified substring</td>
</tr>
<tr>
<td>insertBefore</td>
<td>inserts string before specified substring</td>
</tr>
<tr>
<td>pad</td>
<td>adds leading or trailing characters to strings</td>
</tr>
<tr>
<td>strip</td>
<td>removes leading and trailing characters from string</td>
</tr>
<tr>
<td>lower</td>
<td>converts string to lowercase</td>
</tr>
<tr>
<td>upper</td>
<td>converts string to uppercase</td>
</tr>
<tr>
<td>reverse</td>
<td>reverses order of characters in string</td>
</tr>
<tr>
<td>deblank</td>
<td>removes trailing whitespace from end of string or character array</td>
</tr>
<tr>
<td>strtrim</td>
<td>removes leading and trailing whitespace from string array or character array</td>
</tr>
<tr>
<td>strjust</td>
<td>justifies string or character array</td>
</tr>
</tbody>
</table>

#### Find and Replace

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td></td>
</tr>
<tr>
<td>contains</td>
<td>determines if pattern is in string</td>
</tr>
<tr>
<td>count</td>
<td>counts occurrences of pattern in string</td>
</tr>
<tr>
<td>endsWith</td>
<td>determines if string ends with pattern</td>
</tr>
<tr>
<td>startsWith</td>
<td>determines if string starts with pattern</td>
</tr>
<tr>
<td>strfind</td>
<td>finds one string within another</td>
</tr>
<tr>
<td>sscanf</td>
<td>Reads formatted data from string</td>
</tr>
<tr>
<td>Replace</td>
<td></td>
</tr>
<tr>
<td>replace</td>
<td>finds and replaces substrings in string array</td>
</tr>
<tr>
<td>replaceBetween</td>
<td>replaces substrings identified by indicators that mark their starts and ends</td>
</tr>
<tr>
<td>strrep</td>
<td>finds and replaces substring</td>
</tr>
</tbody>
</table>
## Join and Split

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>join</strong></td>
<td>combines strings</td>
</tr>
<tr>
<td><strong>split</strong></td>
<td>splits strings in string array</td>
</tr>
<tr>
<td><strong>splitlines</strong></td>
<td>splits string at newline characters</td>
</tr>
<tr>
<td><strong>strjoin</strong></td>
<td>joins text in array</td>
</tr>
<tr>
<td><strong>strsplit</strong></td>
<td>splits string at specified delimiter</td>
</tr>
<tr>
<td><strong>strtok</strong></td>
<td>selects parts of string using specified delimiters</td>
</tr>
</tbody>
</table>

## Compare

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>strcmp</strong></td>
<td>compares strings</td>
</tr>
<tr>
<td><strong>strcmpi</strong></td>
<td>compares strings (case insensitive)</td>
</tr>
<tr>
<td><strong>strncmp</strong></td>
<td>compares first $n$ characters of strings (case sensitive)</td>
</tr>
<tr>
<td><strong>strncmpi</strong></td>
<td>compares first $n$ characters of strings (case insensitive)</td>
</tr>
</tbody>
</table>

## Regular Expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>regexp</strong></td>
<td>matches regular expression (case sensitive)</td>
</tr>
<tr>
<td><strong>regexpi</strong></td>
<td>matches regular expression (case insensitive)</td>
</tr>
<tr>
<td><strong>regexprep</strong></td>
<td>replaces text using regular expression</td>
</tr>
<tr>
<td><strong>regexpretranslate</strong></td>
<td>translates text into regular expression</td>
</tr>
</tbody>
</table>

If needed, `matlab.lang.makeValidName(S)` could help you constructing a valid MATLAB variable name from an input string $S$.

Text Analytics Toolbox provides algorithms and visualizations for preprocessing, analyzing, and modeling text data.
for i=1:5
    v=['Var' int2str(i) 'Pts'];
%    v=genvarname(['Var' int2str(i) 'Pts']); % checks name’s validity
    eval([v ' = ' 'i'' ';']);
end

Creating the Variable Names

In the future releases genvarname will be replaced with matlab.lang.makeValidName and matlab.lang.makeUniqueStrings
The End of Chapter 3

Questions?