Plotting Styles

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star		
y	yellow	s	square		
k	black	d	diamond		
		v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

```
% usage: plot( ... , 'b*--')
```

Random Numbers

```
rand([M, N, P], 'datatype'); % uniformly distributed random numbers between 0 & 1
randn([M, N, P], 'datatype'); % normally distributed random numbers
(b - a) * rand + a; % random number between a & b (can use with either of the two functions shown above)
randi([a, b], [M, N, P]); % uniformly distributed random integers between a & b
```

Data Types

Name	Description	Range	Fractions?
logical	representing false and true	0 & 1	no
uint8	unsigned 8-bit integers	0 ... 2^8	no
int8	signed 8-bit integers	-2^8 ... 2^8	no
single	single precision "real" numbers	-realmax ... realmax	yes
double	double precision "real" numbers	-realmax ... realmax	yes

```
% 16, 32, 64-bit also available for unsigned/signed int.
```

Operators and Special Characters

Arithmetic Operators

```
MATLAB uses standard mathematical symbols: +, -, *, /, ^
For element-wise operations, use '.' before the mathematical operator
```

Relational Operators

Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Logical Operators

Symbol	Role
&	logical AND
|	logical OR
~	logical NOT

Special Characters

Symbol	Role
,	Separator for row elements
:	Indexing all elements in list, also used for vector creation
;	Separator for column elements
( )	Operator Precedence
[ ]	Array creation, multiple output argument assignment

```
%      | Comment
""     | String constructor
~      | Argument placeholder (suppress specific output)
=      | Assignment
```

Special Arrays

```
zeros(M, N); % 0 array
false(M, N); % logical false array
```

Array Comparisons

```
A = rand(M, N); % random array
mask = A > 0.5; % logical array where TRUE if >0.5 and FALSE if <=0.5
```

Other Functions

```
who -file <filename>; % List variables in .mat file
pause(1); % Pause script for 1 second
```

Image Processing

Finding Area

```
f = figure;
imshow('file.png');
p = drawpolygon(f.Children); % trace polygon
coords = p.Position;
x_coords = coords(:, 1); % x-coordinates of points
y_coords = coords(:, 2); % y-coordinates of points
area_px_2 = polyarea(x_coords, y_coords); % area of desired object [px^2]
l = drawline(f.Children); % trace scaler bar
length_px = sqrt((l.Position(2,1)-l.Position(1,1))^2+(l.Position(2,2)-
l.Position(1,2))^2); % length of scale bar in [px]
m_per_px = actual_scale_length / length_px; % [m] per [px]
m_per_px_2 = (meters_per_pixel^2) * area_px_2; % area of desired object [m^2]
```

Geolocation

```
longitudes = [153.02];
latitudes = [-27.46];
origin = [mean(longitudes), mean(latitudes)]; % arbitrary origin
radius = 6373.6; % radius of Earth
circumference = 2 * pi * radius;
km_per_degree_latitude = circumference / 360;
km_per_degree_longitude = km_per_degree_latitude * cos(deg2rad(-27.5)); % near
Brisbane
% coordinates to plot
x = (longitudes - origin(1)) * km_per_degree_longitude;
y = (latitudes - origin(2)) * km_per_degree_latitude;
```

Images from Array

Random Greyscale Image

```
A = randi([0, 255], M, N, 'uint8');
```

Random Colour Image

```
A = randi([0 255], M, N, 3, 'uint8');
```

Create Colour Image using Array Indexing

```
A = 255 * ones(M, N, 3, 'uint8'); % white image
A(:, :, 1) = r;
A(:, :, 2) = g;
A(:, :, 3) = b;
% change specific regions using array indexing
A(a:b, c:d, 1) = r;
A(a:b, c:d, 2) = g;
A(a:b, c:d, 3) = b;
```

Editing an Image

```
image = imread('image.png');
% mask certain colour range which can be modified
mask = image(:, :, 1) > r & image(:, :, 2) > g & image(:, :, 3) > b;
% channels 1 2 3 are Red/Green/Blue or Colour/Saturation/Value respectively
imshow(A); % Display Image
image(A); % Similar functionality, useful when used in combination with other plots
```

Save an Animation

```
f = figure;
set(f, 'Visible', 'on')
video = VideoWriter('file_name.avi');
x = []; % x-values
y = []; % y-values
plot1 = plot(x(1), y(1)); % Create plot object
for i = 1:length(t)
    % Update plot object data
    plot1.XData = x(i);
    plot1.YData = y(i);
    drawnow
    frame = getframe;
    writeVideo(video, frame);
end
close(video); % Close video object
```

Sound Processing

```
f = 523.251;                % frequency of note
Fs = 8192;                  % sampling rate
t = 0:1 / Fs:1;             % length of sound
y = sin(2 * pi * f * t);    % sine wave of sound
Y = [sound1a + sound1b];    % play sounds simultaneously, (must be same dimension)
Y = [sound1; sound2];       % append sounds
soundsc(y, Fs)              % play sound (‘sc’ scales between -1 & 1)
resample(y, Fs, Q);         % resample audio at Fs/Q sample rate
Fs / 2;                     % half speed
Fs * 2;                     % double speed
% useful formulas/conversions
duration = length(y) / Fs;
t = linspace(0, duration, length(y));
plot(t, y, '.-'); ylim([-1, 1]);
audiowrite('music.wav', y, Fs) % write sound to audio file
```

Random walks

Initialisation

```
M = 10000;          % number of particles
N = 200;            % number of steps
Delta = 1;          % size of the steps
p = 0.5;            % probability of jumping left
x = zeros(N+1, M);  % initialise particles at 0
```

Computation

```
for i = 1:N
    r = rand(1, M); % random number for each particle
    left_mask = r < p; % mask left-moving particles
    x(i + 1, left_mask) = x(i, left_mask) - Delta; % move them left
    right_mask = ~left_mask; % mask right-moving particles
    x(i + 1, right_mask) = x(i, right_mask) + Delta; % move them right
```

```
end
```
Plot position vs step graph
```
f = figure;
plot(x, '.-');
xlabel('Step number n');
ylabel('Position x_n');
```
Animate positions
```
f = figure;
set(f, 'Visible', 'on');
plot1 = plot(x(1, :), zeros(1, M), '.', 'MarkerSize', 20)
L = max(abs(x(:)));
xlim([-L, L]);
for i = 1:N
    plot1.XData = x(i, :);
end
```
Cellular automata
Initialisation
```
N = 50;     % number of steps
C = 100;    % number of cells
A = false(N + 1, C); % Empty logical array that will contain each iteration
A = rand(1, C) > 0.5; % use random initial state
% manually set initial state (same length as C)
% A(1, :) = [0 1 0 1 1 1 0 0 1];
```
Computation
```
for i = 1:N
    % Arrays of centre, left and right neighbours for the current iteration
    P = A(i, :);
    % Wrap-around boundary cell as ghost cell
    L = [P(C), P(1:C - 1)];
    R = [P(2:C), P(1)];
    % Dead ghost cell
    L = [0, P(1:C - 1)];
    R = [P(2:C), 0];
    % Logical arrays of all possible configurations
    C000 = (L == 0 & P == 0 & R == 0);
    C001 = (L == 0 & P == 0 & R == 1);
    C010 = (L == 0 & P == 1 & R == 0);
    C011 = (L == 0 & P == 1 & R == 1);
    C100 = (L == 1 & P == 0 & R == 0);
    C101 = (L == 1 & P == 0 & R == 1);
    C110 = (L == 1 & P == 1 & R == 0);
    C111 = (L == 1 & P == 1 & R == 1);
    % Determine the logical mask for our simulation
    live_mask = C001 | C011 | C010 | C100; % Rule for cellular automation
    % Set live cells for next iteration
    A(i + 1, live_mask) = 1;
end
imshow(~A, 'InitialMagnification', 'Fit') % Using NOT as live cells are black
```