FF505/FY505

Computational Science

Lecture 3

Programming: Control Flow Functions, Graphics

Marco Chiarandini (marco@imada.sdu.dk)

Department of Mathematics and Computer Science (IMADA)
University of Southern Denmark

Outline

1. Exercise: Monte Carlo Simulation
   - Improving Performance
2. Programming
3. Functions
   - Exercise
4. Graphics
   - 2D Plots
   - 3D Plots

Resume

- Overview of MATLAB environment
- Overview of MATLAB programming and arrays
- Linear Algebra in MATLAB
  (Matrix and element-by-element operations)
- Solving linear systems in MATLAB

Today

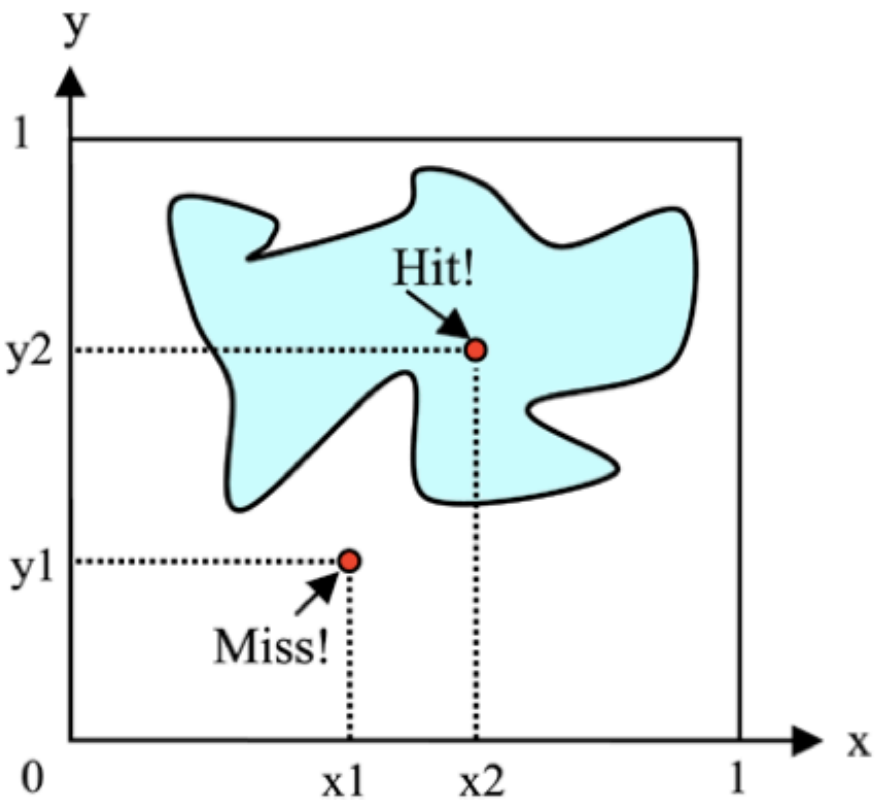
- Programming: Control structures
- Writing your own Functions
- Graphics: basic and advanced plotting
- Efficiency issues

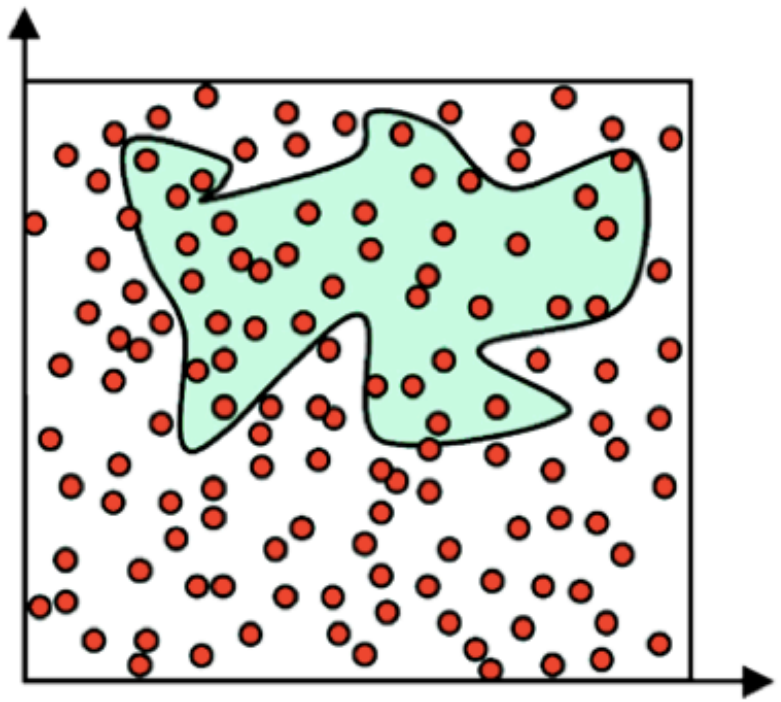
Outline

1. Exercise: Monte Carlo Simulation
   - Improving Performance
2. Programming
3. Functions
   - Exercise
4. Graphics
   - 2D Plots
   - 3D Plots

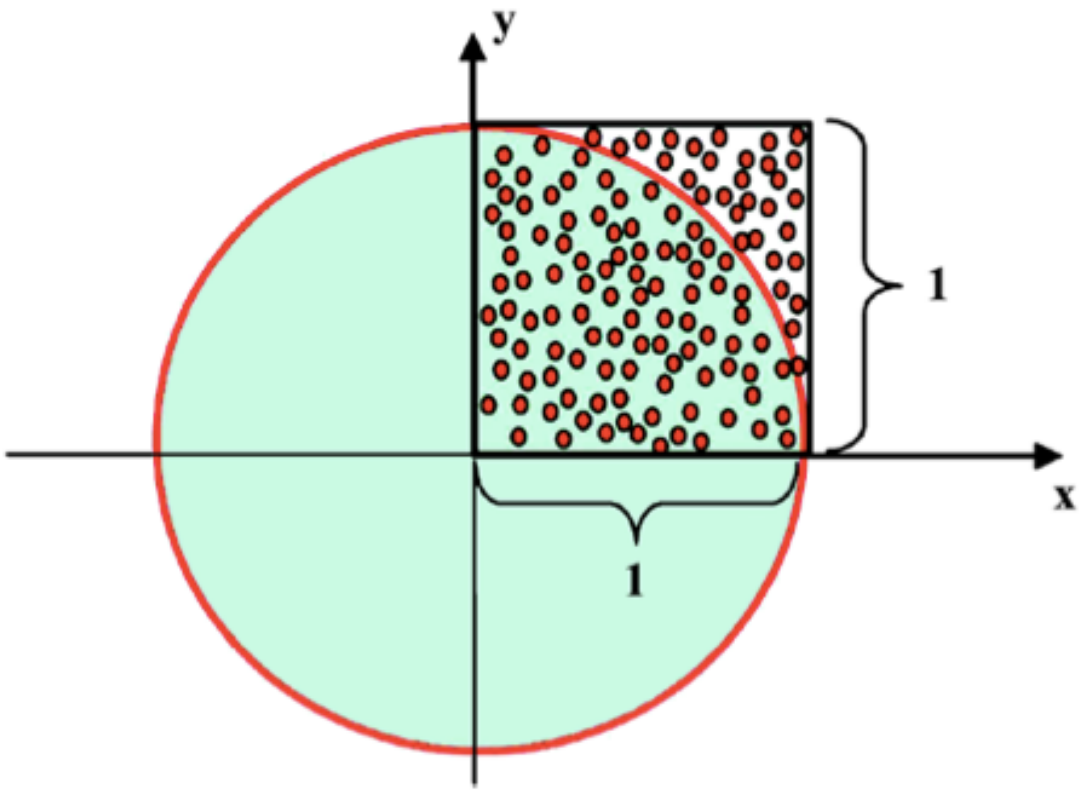
Monte Carlo Simulation

Calculate area by random rain:





Calculate π



Solution

Let A_s be the simulated area:

$$\frac{\pi}{4} = A_s$$

```
S=1000;
hits = 0;
for k = 1:S
    x = rand(1);
    y = rand(1);
    P = x^2+y^2;
    hits = P<1;
end
As=hits/S;
pi=4*As;
```

```
S=1000;
XY=rand(S,2);
P=sum(XY.^2,2);
hits=sum(P<1);
As=hits/S;
pi=4*As;
```

Script and Function Files (M-Files)

Script file

```
x=(1:1000)';
for k=1:5
  y(:,k)=k*log(x);
end
plot(x,y)
```

command line `simple`
does not take arguments
operates on data in the workspace

Function file

```
function y=simple(maxLoop)
        % (smart indent)
        x=(1:1000)';
        for k=1:maxLoop
        y(:,k)=k*log(x);
        end
        plot(x,y)
```

command line `g=simple(10)`
can take input arguments and return output arguments.
Internal variables are local to the function

Same name conventions for `.m` files as for variables.
Check if variables or functions are already defined.

```
exist("example1")
exist("example1.m","file")
exist("example1","builtin")
```

```
type fun
```

Script and Function Files (M-files)

- Modularize
- Make interaction clear
  make functions interact via arguments (in case structures) rather than via global variables
- Partitioning
- Use existing functions
  (http://www.mathworks.com/matlabcentral/fileexchange)
- Any block of code appearing in more than one m-file should be considered for packaging as a function
- Subfunctions
  packaged in the same file as their functions
- Test scripts

Efficient Code

```
function mypi=calculate_pi_1(S)
   hits = 0;
   for k = 1:S
       x = rand(1);
       y = rand(1);
       P = x^2+y^2;
       hits = P<1;
   end
   As=hits/S;
   mypi=4*As;
```

```
function mypi=calculate_pi_2(S)
   S=1000;
   XY=rand(S,2);
   P=sum(XY.^2,2);
   hits=sum(P<1);
   As=hits/S;
   mypi=4*As;
```

```
tic,
for k=1:100
   calculate_pi_1(1000);
end
toc
```

```
tic,
for k=1:100
   calculate_pi_2(1000);
end
toc
```

Techniques for Improving Performance

Can you improve performance and use memory more efficiently for this code?

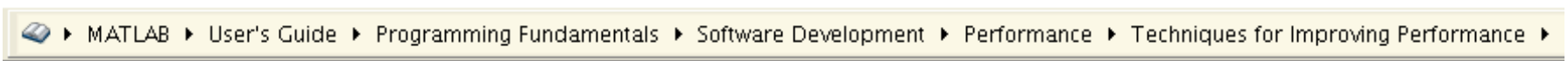
```
A=rand(1000,400)>0.7
s=[]
M=0
for j=1:400
    tmp_s=0
    for i=1:1000
        if A(i,j)>M
            M=A(i,j)
        end
        if A(i,j)>0
            tmp_s=tmp_s+A(i,j)
    end
    s=[s, tmp_s]
end
```

Use `tic ... toc` and `whos` to analyse your code.

`tic; bad; toc`

For inspiration look at User's Guide:

MATLAB > User's Guide > Programming Fundamentals > Software Development > Performance > Techniques for Improving Performance



Outline

1. Exercise: Monte Carlo Simulation
   - Improving Performance
2. Programming
3. Functions
   - Exercise
4. Graphics
   - 2D Plots
   - 3D Plots

Algorithms and Control Structures

Algorithm: an ordered sequence of instructions that perform some task in a finite amount of time.

Individual statements, instructions or function calls can be numbered and executed in sequence, but an algorithm has the ability to alter the order of its instructions. The order is referred to as control flow.

Three categories of control flow:

- Sequential operations
- Conditional operations: logical conditions that determine actions.
- Iterative operations (loops)

For an imperative or a declarative program a control flow statement is a statement whose execution results in a choice being made as to which of two or more paths should be followed.

For non-strict functional languages (like Matlab), functions and language constructs exist to achieve the same result, but they are not necessarily called control flow statements (eg, vectorization).

Relational Operators

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

```
islogical(5~=8)
ans =
     1
islogical(logical(5+8))
ans =

     1
>> logical(5+8)
ans =
     1
>> double(6>8)
ans =
     0
>> isnumeric(double(6>8))
ans =
     1
```

Logical Operators

`~`	NOT	`~A` returns an array the same dimension as A; the new array has ones where A is zero and zeros where A is nonzero.
`&`	AND	`A & B` returns an array the same dimension as A and B; the new array has ones where both A and B have nonzero elements and zeros where either A or B is zero.
`|`	OR	`A | B` returns an array the same dimension as A and B; the new array has ones where at least one element in A or B is nonzero and zeros where A and B are both zero.
`&&`	Short-Circuit AND	Operator for scalar logical expressions. `A && B` returns true if both A and B evaluate to true, and false if they do not.
`||`	Short-Circuit OR	Operator for scalar logical expressions. `A || B` returns true if either A or B or both evaluate to true, and false if they do not.

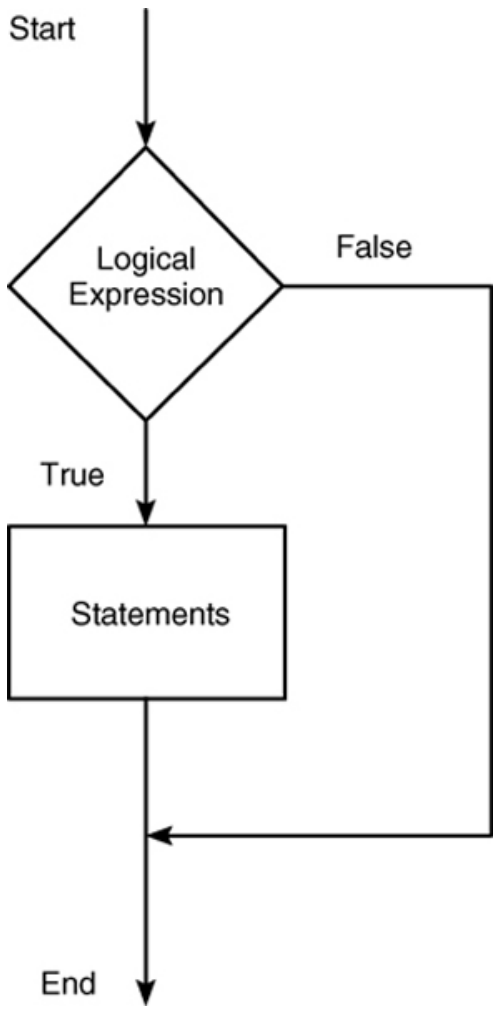
Precedence

1. Parentheses; evaluated starting with the innermost pair.
2. Arithmetic operators and logical NOT (~); evaluated from left to right.
3. Relational operators; evaluated from left to right.
4. Logical AND.
5. Logical OR.

The `if` Statement

The if statement's basic form is

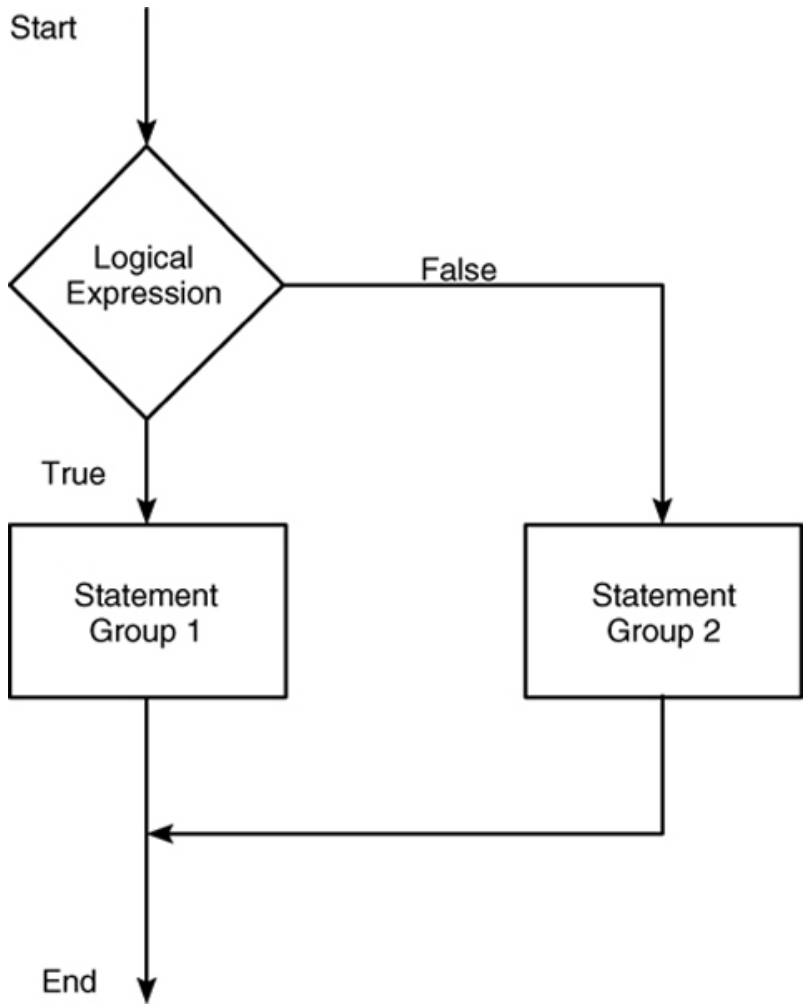
```
if logical expression
     statements
end
```



The `else` Statement

The basic structure for the use of the else statement is

```
if logical expression
     statement group 1
else
     statement group 2
end
```



```
if logical expression 1
   if logical expression 2
        statements
   end
end
```

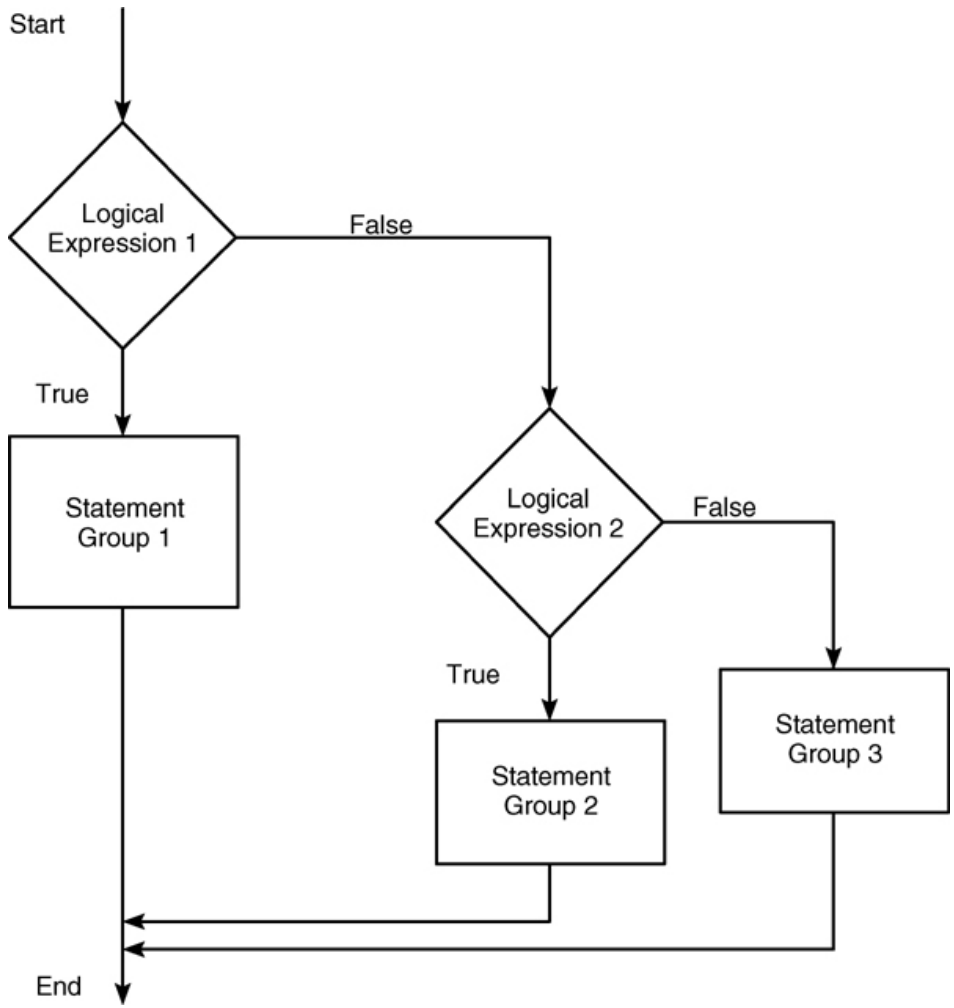
can be replaced with the more concise program

```
if logical expression 1 & logical expression 2
     statements
end
```

The `elseif` Statement

The general form of the if statement is

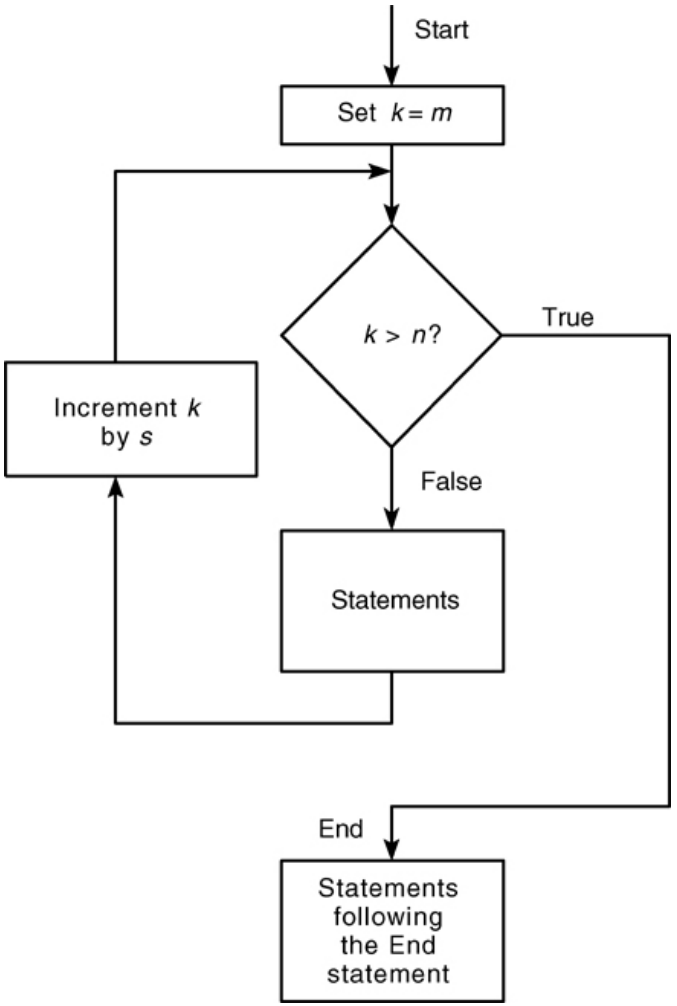
```
if logical expression 1
    statement group 1
elseif logical expression 2
    statement group 2
else
    statement group 3
end
```



for Loops

A simple example of a for loop is

```
for k = 5:10:35
    x = k^2
end
```

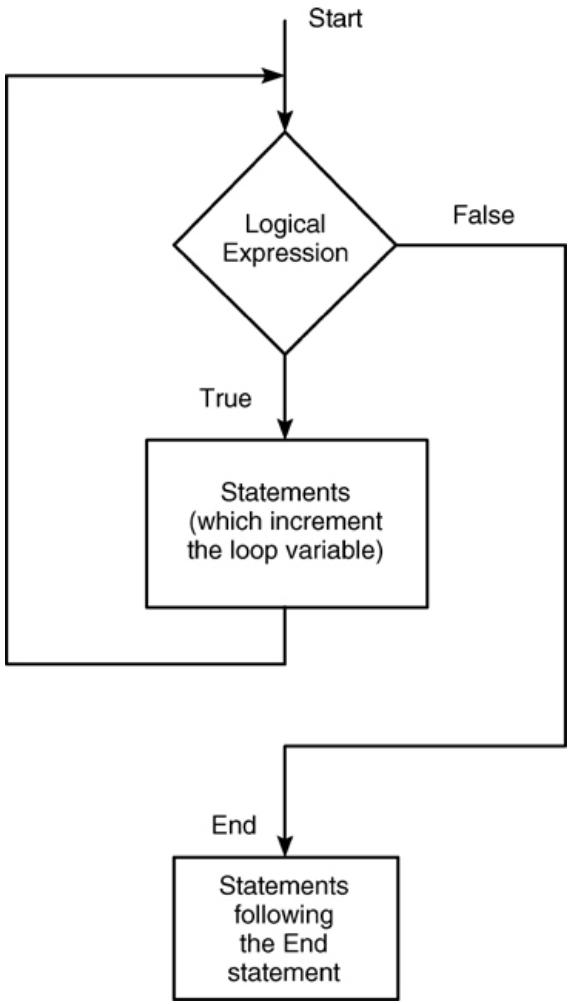


`while` Loops

```
while logical expression
    statements
end
```

The while loop is used when the looping process terminates because a specified condition is satisfied, and thus the number of passes is not known in advance.

```
x = 5;
while x < 25
    disp(x)
    x = 2*x - 1;
end
```



switch

```
switch input expression % (can be a
      scalar or string).
   case value1
       statement group 1
   case value2
       statement group 2
    .
    .
    .
   otherwise
       statement group n
end
```

```
switch angle
    case 45
      disp('Northeast')
    case 135
      disp('Southeast')
    case 225
      disp('Southwest')
    case 315
      disp('Northwest')
    otherwise
      disp('Direction Unknown')
end
```

Control Flow

if

```
if w(1)==0
    % <statement>
elseif w(1)==1
    % <statement>
else
    % <statement>
end
```

switch

```
method = 'Bilinear';
switch lower(method)
   case {'linear','bilinear'}
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
   case 'nearest'
      disp('Method is nearest')
   otherwise
      disp('Unknown method.')
end
```

for

```
w = [];
z = 0;
is = 1:10
for i=is
    w = [w, 2*i] % Same as \/
% w(i) = 2*i
% w(end+1) = 2*i
    z = z + i;
    % break;
    % continue;
end
% avoid! same as w = 2*[1:10], z = sum([1:10]);
```

while

```
w = [];
while length(w) < 3
    w = [w, 4];
    % break
end
```

Continue and Break

The continue statement passes control to the next iteration of the for loop or while loop in which it appears, skipping any remaining statements in the body of the loop.
The break statement is used to exit early from a for loop or while loop. In nested loops, break exits from the innermost loop only.

This will never end

```
while count <= 20
    if true
        continue
    end
    count = count + 1;
end
```

This will iterate once and stop

```
while count <= 20
    if true
        break
    end
    count = count + 1;
end
```

Vectorization

MATLAB is optimized for operations involving matrices and vectors.
Vectorization: The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations

A simple example to create a table of logarithms:

loop-based, scalar-oriented code:

```
x = .01;
for k = 1:1001
   y(k) = log10(x);
   x = x + .01;
end
```

A vectorized version of the same code is

```
x = .01:.01:10;
y = log10(x);
```

Some functions are vectorized, hence with vectors must use element-by-element operators to combine them.
Eg: $z = e^y \sin x$, x and y vectors:

```
z=exp(y).*sin(x)
```

Vectorization

Vectorizing your code is worthwhile for:

- Appearance: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- Less Error Prone: Without loops, vectorized code is often shorter. Fewer lines of code mean fewer opportunities to introduce programming errors.
- Performance: Vectorized code often runs much faster than the corresponding code containing loops.

Preallocation

Another speedup techinque is preallocation. Memory allocation is slow.

```
r = zeros(32,1);
for n = 1:32
    r(n) = rank(magic(n));
end
```

Without the preallocation MATLAB would enlarge the r vector by one element each time through the loop.

Outline

1. Exercise: Monte Carlo Simulation
   Improving Performance

2. Programming

3. Functions
   Exercise

4. Graphics
   2D Plots
   3D Plots

User-Defined Functions

The first line in a function file distinguishes a function M-file from a script M-file. Its syntax is as follows:

```
function [output variables] = name(input variables)
```

The function name should be the same as the file name in which it is saved (with the .m extension).

Example

```
function z = fun(x,y)
% the first line of comments is accessed by lookfor
% comments immediately following the definition
% are shown in help
u = 3*x;
z = u + 6*y.^2;
```

```
q = fun(3,7)
q =
   303
```

⇝ variables have local scope

Variable Scope

Local Variables: do not exist outside the function.

```
function z = fun(x,y)
u = 3*x;
z = u + 6*y.^2;
```

```
>> x = 3; y = 7;
>> q = fun(x,y);
>> x
x =
3
>> y
y =
7
>> u
??? Undefined function or variable 'u'.
```

The variables x, y and u are local to the function fun

Local Variables

Local variables do not exist outside the function

```
>>x = 3;y = 7;
>>q = fun(x,y);
>>x
x =
3
>>y
y =
7
>>u
??? Undefined function or variable 'u'.
```

Local Variables

Variable names used in the function definition may, but need not, be used when the function is called:

In fun.m

```
function z = fun(x,y)
x=x+1; %we increment x but x is local and
       will not change globally
z=x+y;
```

At prompt

```
>> x=3;
>> z=fun(x,4)
>> x
x =
     3
```

All variables inside a function are erased after the function finishes executing, except when the same variable names appear in the output variable list used in the function call.

Global Variables

The `global` command declares certain variables global: they exist and have the same value in the basic workspace and in the functions that declare them global.

```
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

```
>> global GRAVITY
>> GRAVITY = 32;
>> y = falling((0:.1:5)');
```

Programming style guidelines recommend avoiding to use them.

Parameters and Arguments

Arguments passed by position Only the order of the arguments is important, not the names of the arguments:

```
>> x = 7; y = 3;
>> z = fun(y, x)
z =
   303
```

The second line is equivalent to z = fun(3,7).

Inside the function variables `nargin` and `nargout` tell the number of input and output arguments involved in each particular use of the function

One can use arrays as input arguments:

```
>> r = fun(2:4,7:9)
r =
   300 393 498
```

A function may have no input arguments and no output list.

```
function show_date
clear
clc
today = date
```

Function Handles

- A function handle is an address to reference a function.
- It is declared via the @ sign before the function name.
- Mostly used to pass the function as an argument to another function.

```
function y = f1(x)
y = x + 2*exp(-x) - 3;
```

```
>> plot(0:0.01:6, @f1)
```

Example: Finding zeros and minima

`X = FZERO(FUN,X0)` system function with syntax:

```
fzero(@function, x0) % zero close to x0
fminbnd(@function, x1, x2) % min between x1 and x2
```

```
fzero(@cos,2)
ans =
    1.5708
>> fminbnd(@cos,0,4)
ans =
    3.1416
```

Ex: plot and find the zeros and minima of $y = x + 2e^x - 3$

To find the minimum of a function of more than one variable

```
fminsearch(@function, x0)
```

where `@function` is a the handler to a function taking a vector and x_0 is a guess vector

Other Ways

```
>> fun1 = 'x.^2-4';
>> fun_inline = inline(fun1);
>> [x, value] = fzero(fun_inline,[0, 3])
```

```
>> fun1 = 'x.^2-4';
>> [x, value] = fzero(fun1,[0, 3])
```

```
>>[x, value] = fzero('x.^2-4',[0, 3])
```

Types of User-Defined Functions

- The primary function is the first function of an M-file. Other are subroutines not callable.
- Subfunctions placed in the file of the primary function, not visible outside the file
- Nested functions defined within another function. Have access to variables of the primary function.
- Anonymous functions at the MATLAB command line or within another function or script

```
% fhandle = @(arglist) expr
>> sq = @(x) (x.^2)
>> poly1 = @(x) 4*x.^2 - 50*x + 5;
>> fminbnd(poly1, -10, 10)
>> fminbnd(@(x) 4*x.^2 - 50*x + 5, -10, 10)
```

- Overloaded functions are functions that respond differently to different types of input arguments.
- Private functions placed in a `private` folder and visible only to parent folder

Function Arguments

Create a new function in a file named addme.m that accepts one or two inputs and computes the sum of the number with itself or the sum of the two numbers. The function must be then able to return one or two outputs (a result and its absolute value).

Outline

1. Exercise: Monte Carlo Simulation
   - Improving Performance
2. Programming
3. Functions
   - Exercise
4. Graphics
   - 2D Plots
   - 3D Plots

Introduction

Plot measured data (points) or functions (lines)

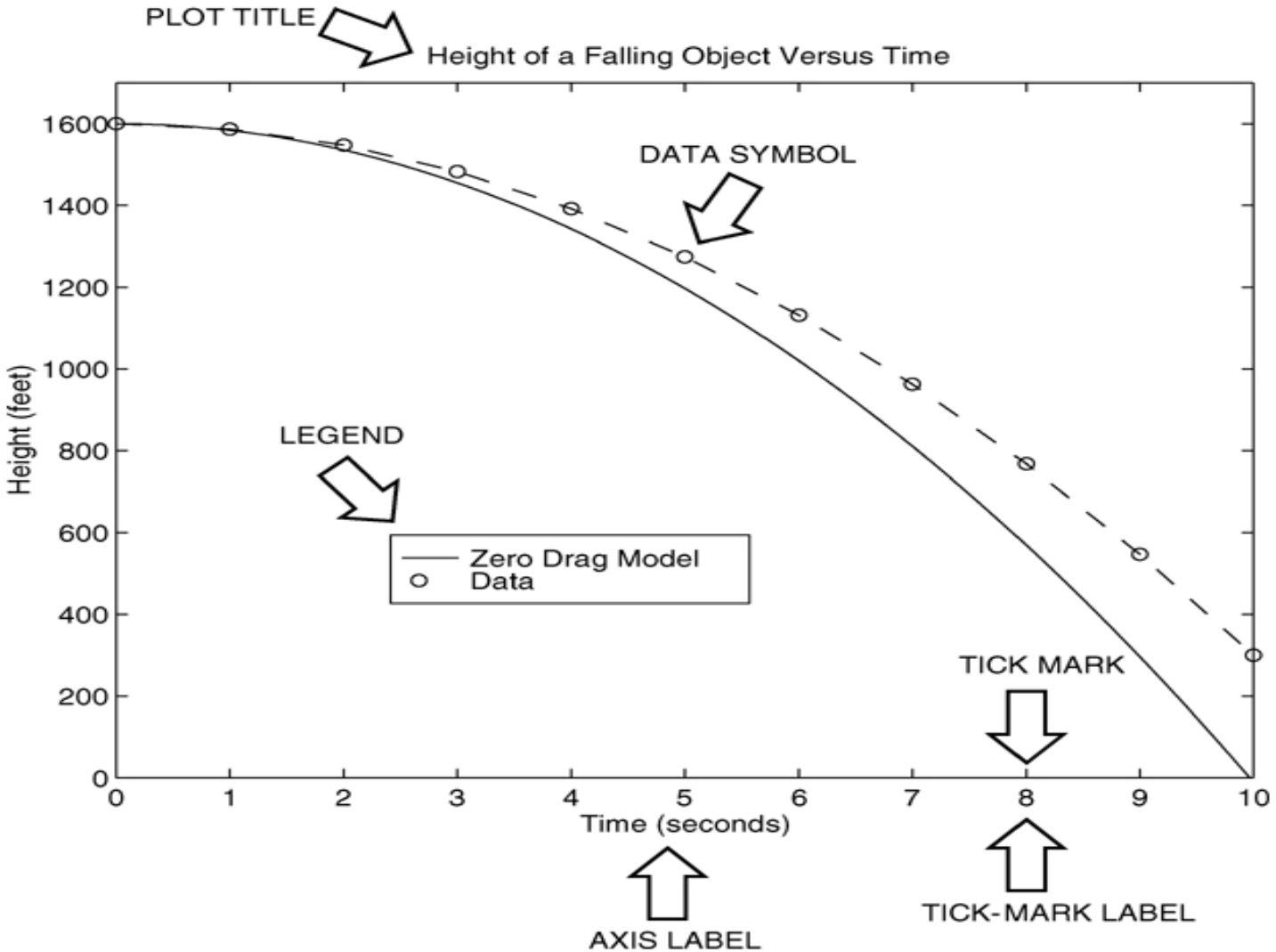
Two-dimensional plots or xy plots

```
help graph2d
```

Three-dimensional plots or xyz plots or surface plots

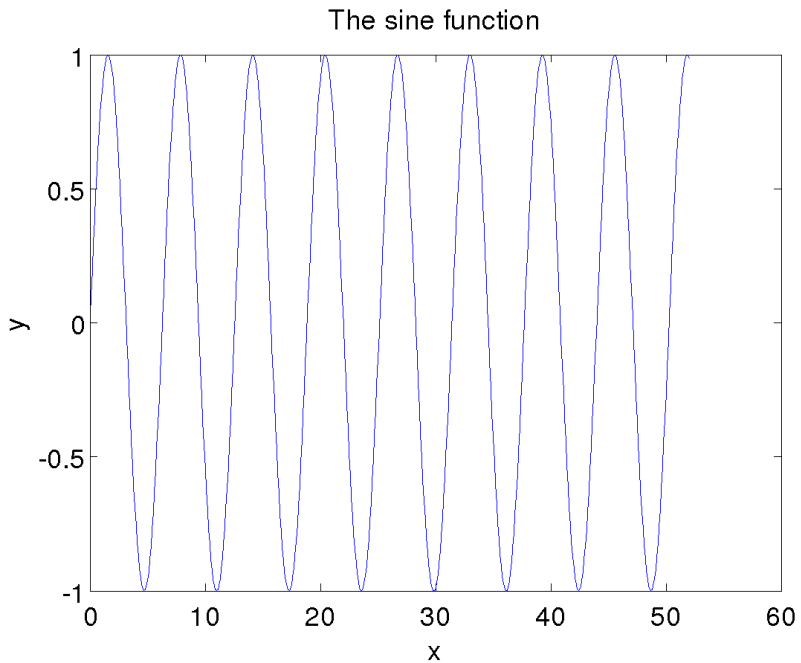
```
help graph3d
```

Nomenclature xy plot



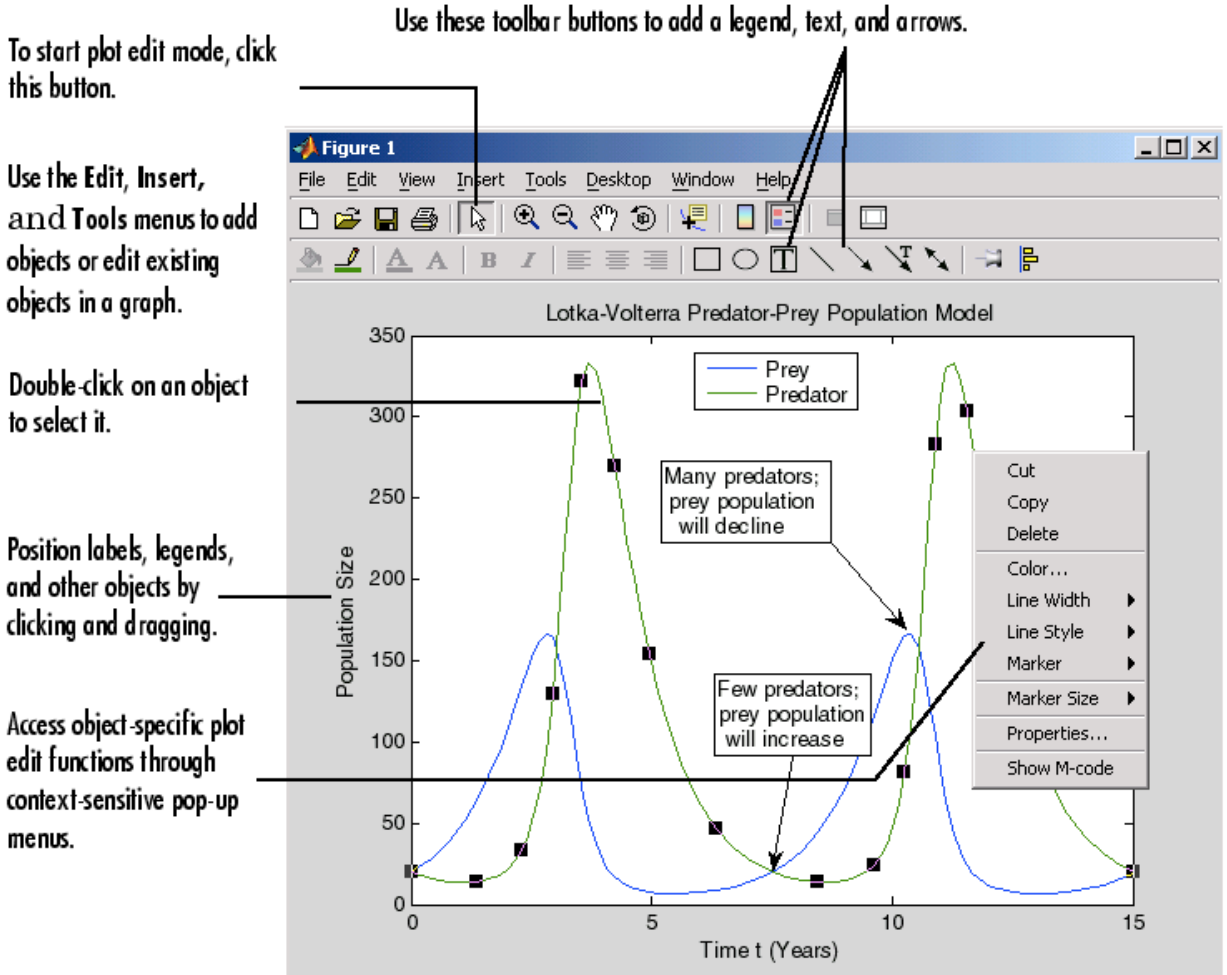
An Example: $y = \sin(x)$

```
x = 0:0.1:52;
y = sin(x)
plot(x,y)
xlabel('x')
ylabel('y')
title('The sine function')
```



The autoscaling feature in MATLAB selects tick-mark spacing.

Plotedit



But better to do this with lines of code, just in case you have to redo the plot.

Saving Figures

The plot appears in the Figure window. You can include it in your documents:

1. type
   `print -dpng foo`
   at the command line. This command sends the current plot directly to `foo.png`

   ⇝ `help print`

2. from the File menu, select Save As, write the name and select file format from Files of Types (eg, png, jpg, etc)
   .fig format is MATLAB format, which allows to edit

3. from the File menu, select Export Setup to control size and other parameters

4. on Windows, copy on clipboard and paste. From Edit menu, Copy Figure and Copy Options

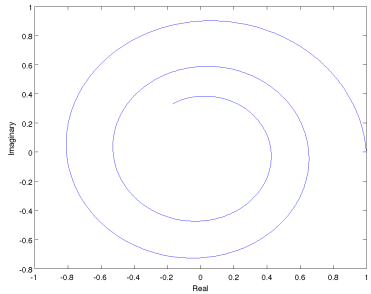
The grid and axis Commands

- `grid` command to display gridlines at the tick marks corresponding to the tick labels.
  `grid on` to add gridlines;
  `grid off` to stop plotting gridlines;
  `grid` to toggle

- `axis` command to override the MATLAB selections for the axis limits.
  `axis([xmin xmax ymin ymax])` sets the scaling for the x- and y-axes to the minimum and maximum values indicated. Note: no separating commas
  `axis square`, `axis equal`, `axis auto`

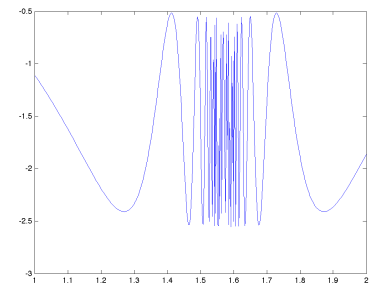
plot complex numbers

```
y=0.1+0.9i, plot(y)
z=0.1+0.9i, n=0:0.01:10,
plot(z.^n), xlabels('Real'), ylabel('Imaginary')
```



function plot command

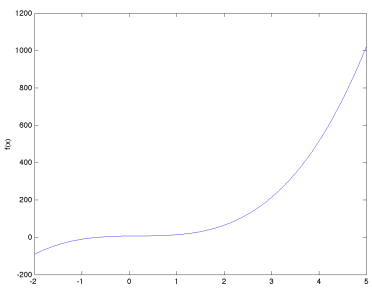
```
f=@(x) (cos(tan(x))-tan(sin(x)));
fplot(f,[1 2])
[x,y]=fplot(function,limits)
```



plotting polynomials

Eg, $f(x) = 9x^3 - 5x^2 + 3x + 7$ for $-2 \leq x \leq 5$:

```
a = [9,-5,3,7];
x = -2:0.01:5;
plot(x,polyval(a,x)),xlabel('x'),ylabel('f(x)')
```

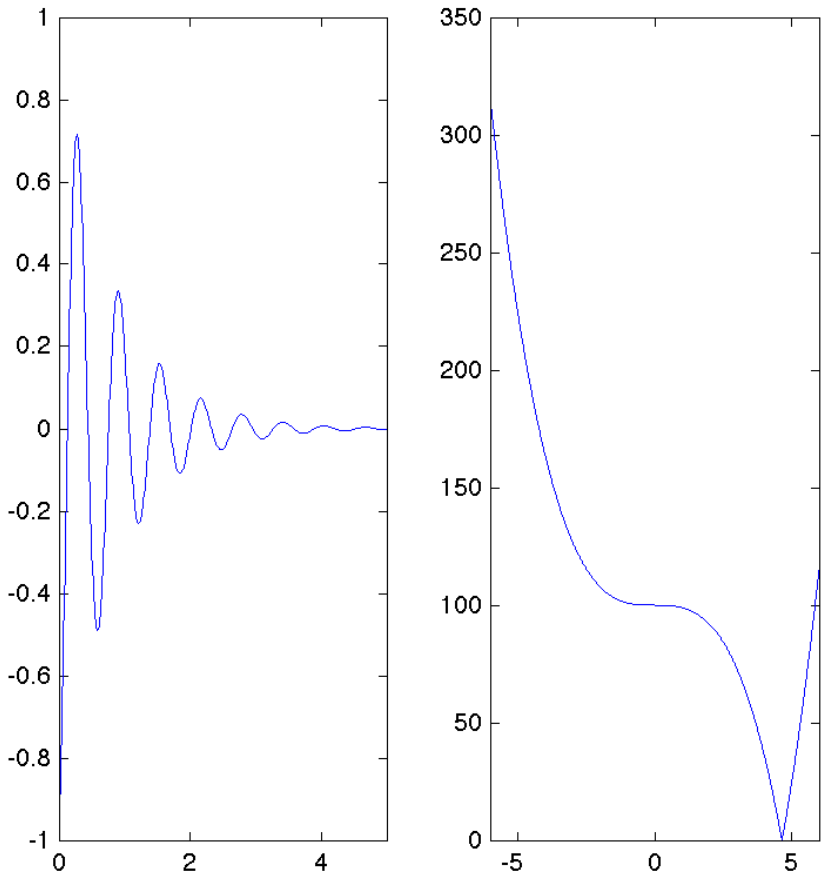


Subplots

`subplot` command to obtain several smaller subplots in the same figure.

`subplot(m,n,p)` divides the Figure window into an array of rectangular panes with m rows and n columns and sets the pointer after the pth pane.

```
x = 0:0.01:5;
y = exp(-1.2*x).*sin(10*x+5);
subplot(1,2,1)
plot(x,y),axis([0 5 -1 1])
x = -6:0.01:6;
y = abs(x.^3-100);
subplot(1,2,2)
plot(x,y),axis([-6 6 0 350])
```



Data Markers and Line Types

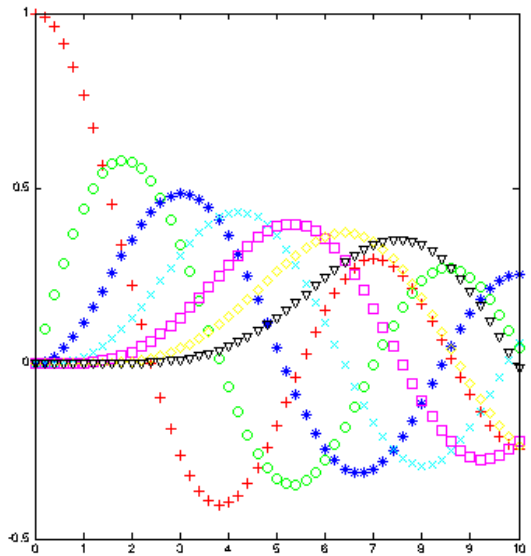
Three components can be specified in the string specifiers along with the plotting command. They are:

- Line style
- Marker symbol
- Color

```
plot(x,y,u,v,'--') % where the symbols '--' represent a dashed line
plot(x,y,'*',x,y,':') % plot y versus x with asterisks connected with a dotted line
plot(x,y,'g*',x,y,'r--') % green asterisks connected with a red dashed line
```

```
% Generate some data using the besselj
x = 0:0.2:10;
y0 = besselj(0,x);
y1 = besselj(1,x);
y2 = besselj(2,x);
y3 = besselj(3,x);
y4 = besselj(4,x);
y5 = besselj(5,x);
y6 = besselj(6,x);

plot(x, y0, 'r+', x, y1, 'go', x, y2, 'b*',
      x, y3, 'cx', ...
   x, y4, 'ms', x, y5, 'yd', x, y6, 'kv');
```



```
doc LineSpec
```

Specifier	LineStyle
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line

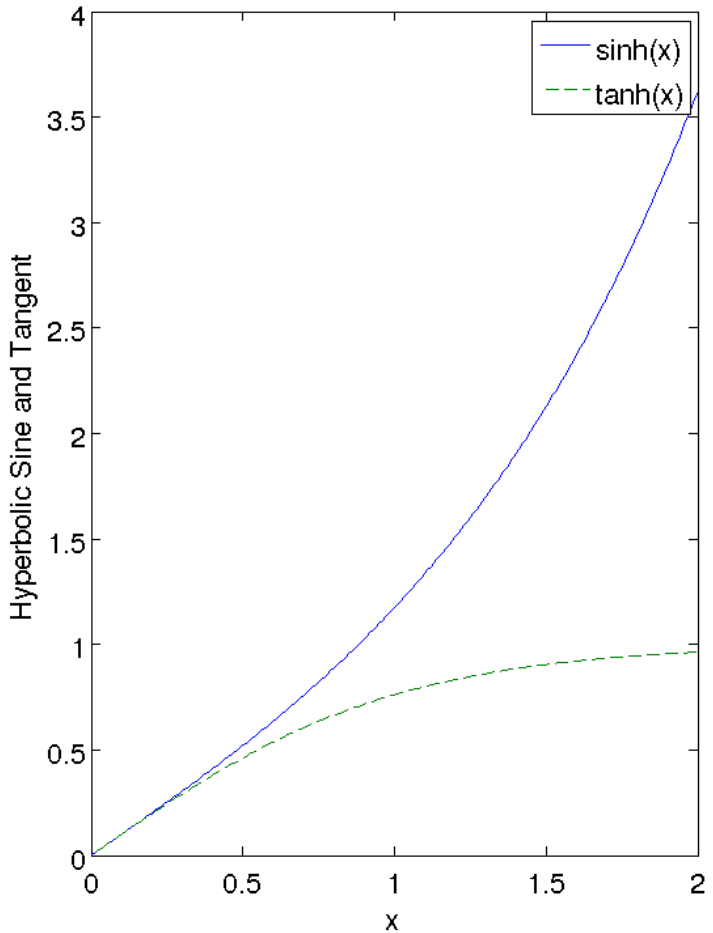
Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h' ''	Six-pointed star (hexagram)

Specifier	Color
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Labeling Curves and Data

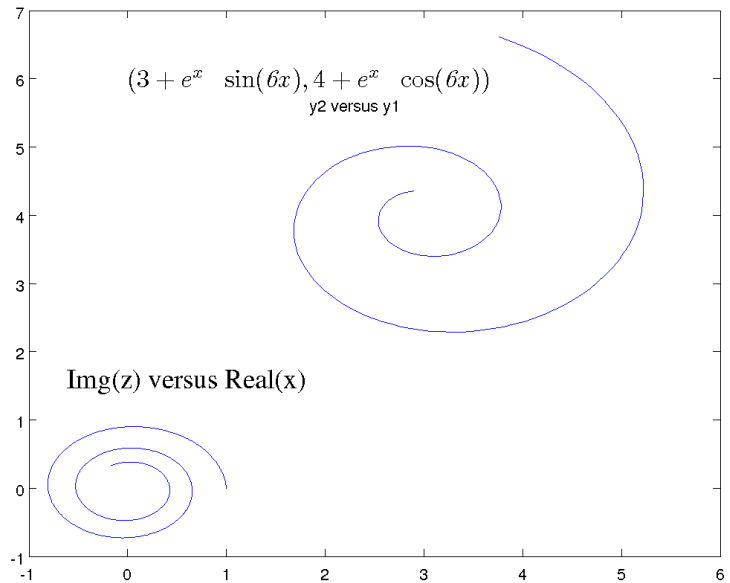
The legend command automatically obtains the line type used for each data set

```
x = 0:0.01:2;
y = sinh(x);
z = tanh(x);
plot(x,y,x,z,'--'),xlabel('x')
ylabel('Hyperbolic Sine and Tangent')
legend('sinh(x)','tanh(x)')
```



The `hold` Command and Text Annotations

```
x=-1:0.01:1
y1=3+exp(-x).*sin(6*x);
y2=4+exp(-x).*cos(6*x);
plot((0.1+0.9i).^(0:0.01:10)), hold, plot(y1,y2)
gtext('y2 versus y1') % places in a point specified by the mouse
gtext('Img(z) versus Real(x)','FontName','Times','Fontsize',18)
```

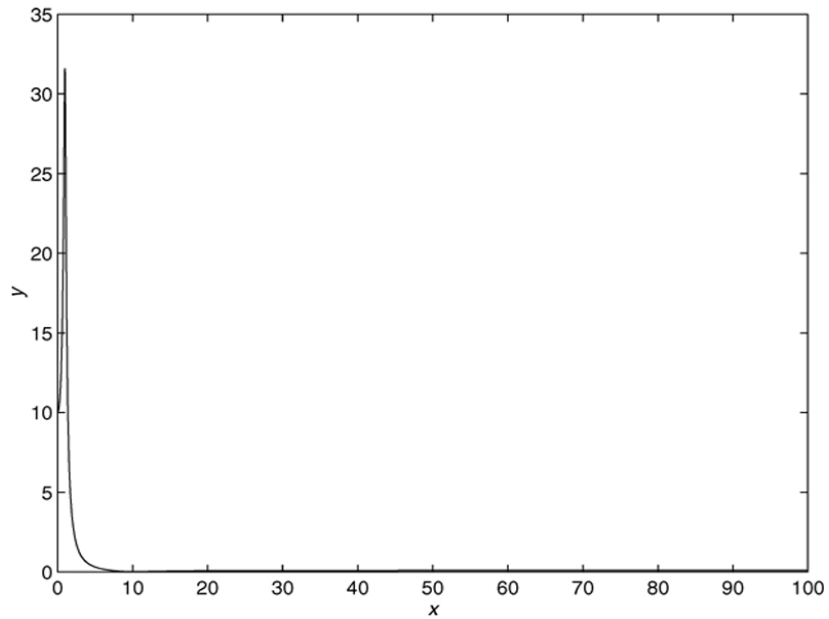


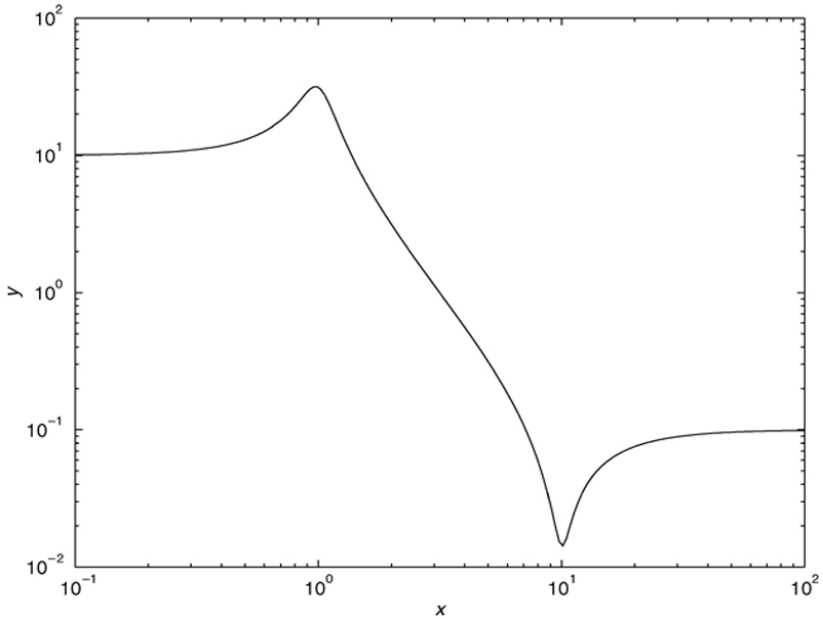

```
text('Interpreter','latex',...
 'String',...
 '$(3+e^{-x}\sin({\it 6x}),4+e^{-x}\cos({\
      it 6x}))$',...
 'Position',[0,6],...
 'FontSize',16)
```

Search Text Properties in Help

Search Mathematical symbols, Greek Letter and TeX Characters

Axes Transformations





Instead of `plot`, plot with

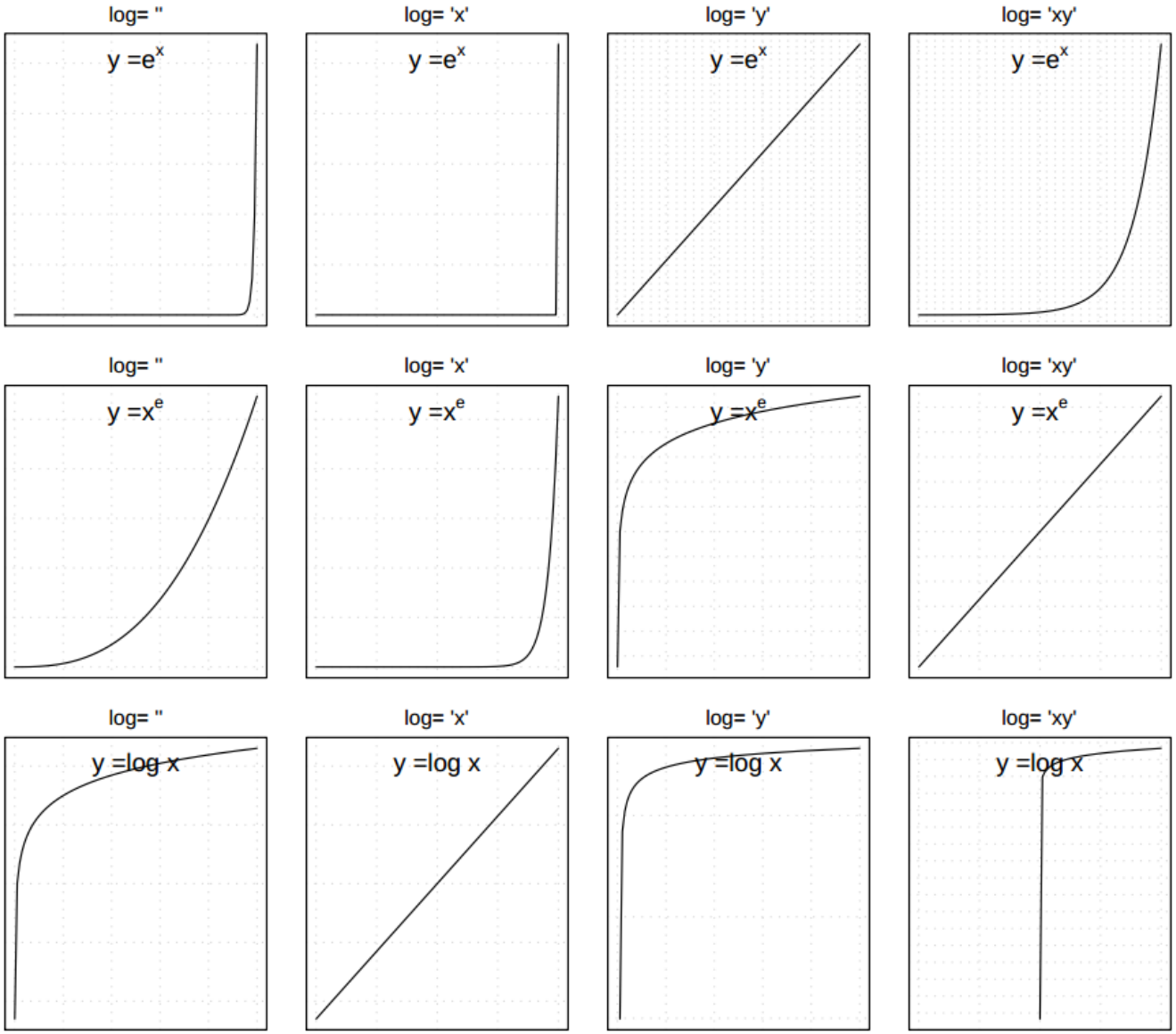
```
loglog(x,y) % both scales logarithmic.
semilogx(x,y) % x scale logarithmic and the y scale rectilinear.
semilogy(x,y) % y scale logarithmic and the x scale rectilinear.
```

Logarithmic Plots

Remember:

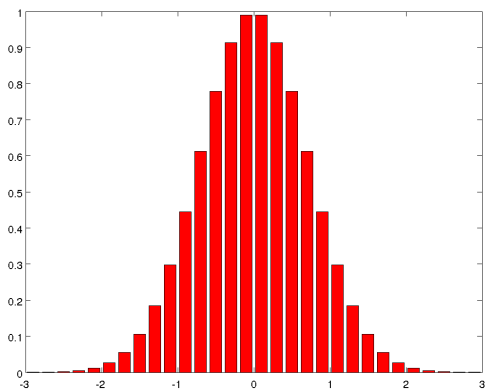
1. You cannot plot negative numbers on a log scale: the logarithm of a negative number is not defined as a real number.
2. You cannot plot the number 0 on a log scale: $\log_{10} 0 = -\infty$.
3. The tick-mark labels on a log scale are the actual values being plotted; they are not the logarithms of the numbers. Eg, the range of x values in the plot before is from $10^{-1} = 0.1$ to $10^2 = 100$.
4. Gridlines and tick marks within a decade are unevenly spaced. If 8 gridlines or tick marks occur within the decade, they correspond to values equal to $2, 3, 4, \ldots, 8, 9$ times the value represented by the first gridline or tick mark of the decade.
5. Equal distances on a log scale correspond to multiplication by the same constant (as opposed to addition of the same constant on a rectilinear scale).

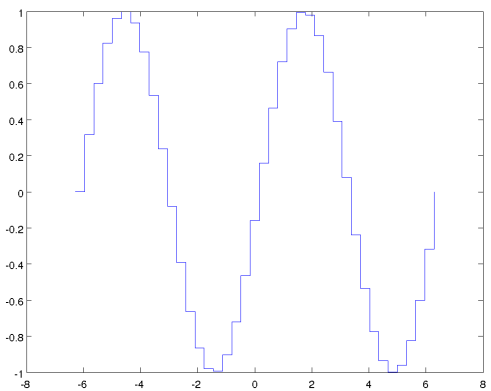
The effect of log-transformation

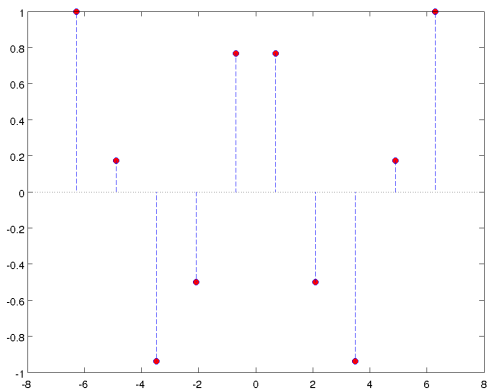


Specialized plot commands

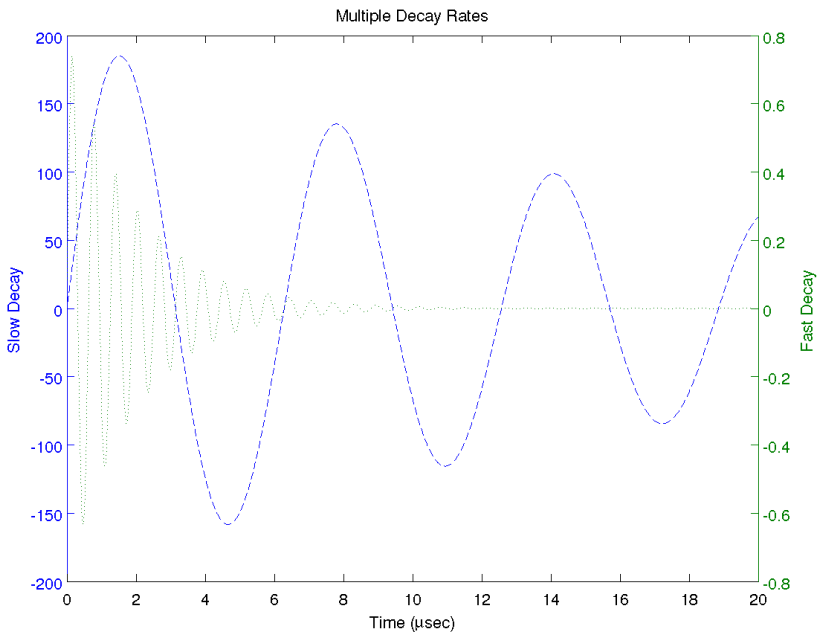
Command	Description
`bar(x,y)`	Creates a bar chart of y versus x
`stairs(x,y)`	Produces a stairs plot of y versus x.
`stem(x,y)`	Produces a stem plot of y versus x.

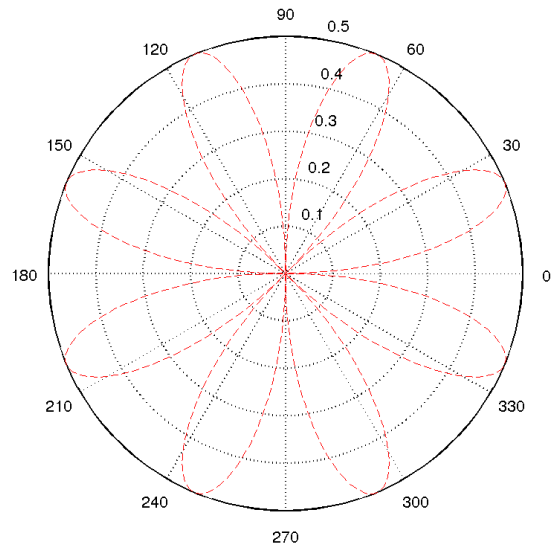






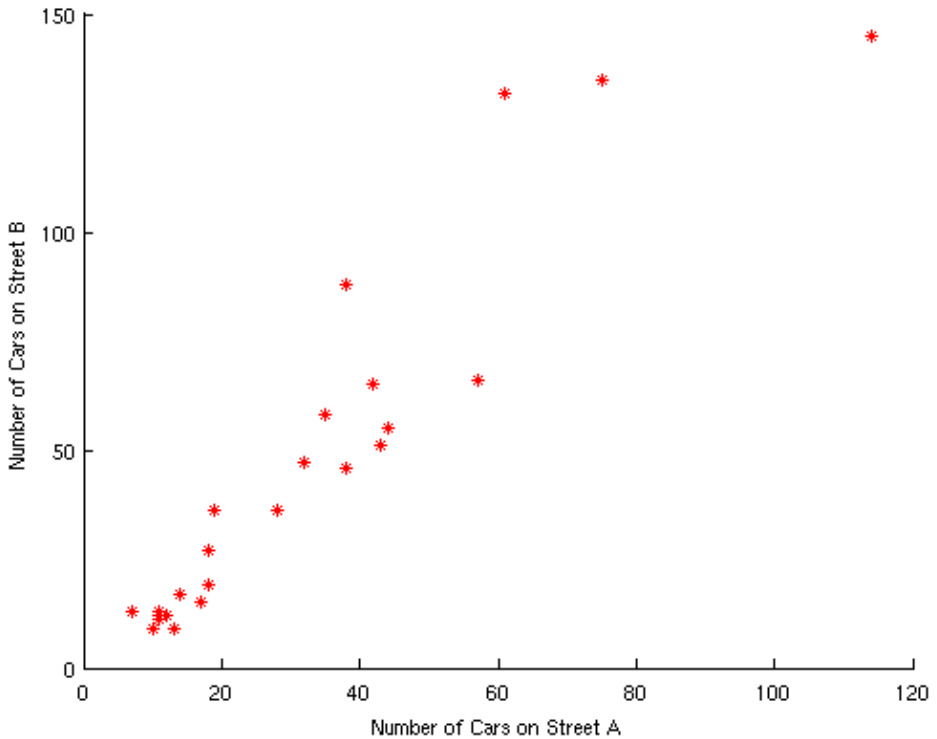
Command	Description
`plotyy(x1,y1,x2,y2)`	Produces a plot with two y-axes, y1 on the left and y2 on the right
`polar(theta,r,'type')`	Produces a polar plot from the polar coordinates theta and r, using the line type, data marker, and colors specified in the string type.





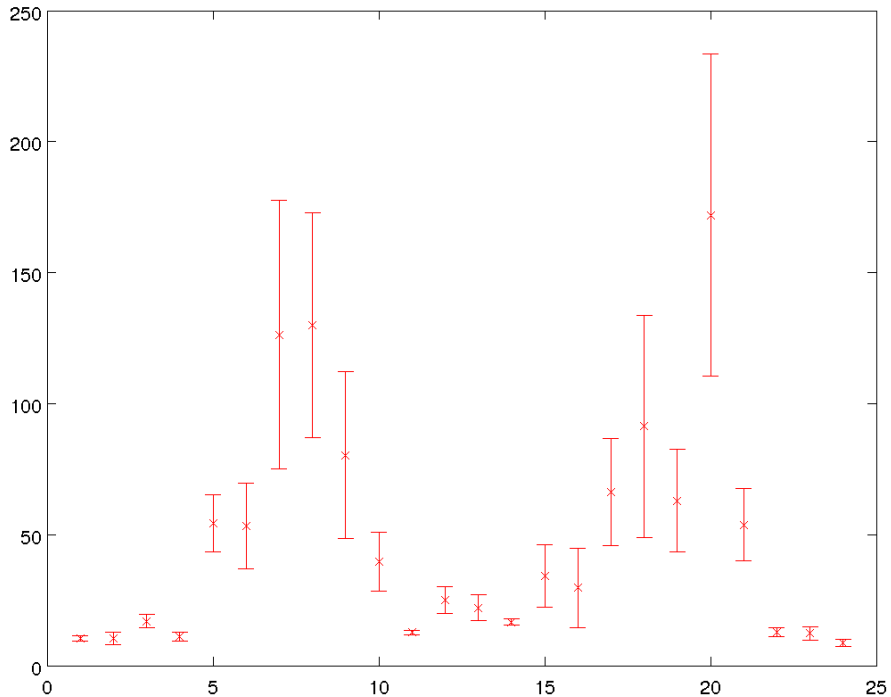
Scatter Plots

```
load count.dat
scatter(count(:,1),count(:,2),
     'r*')
xlabel('Number of Cars on
     Street A');
ylabel('Number of Cars on
     Street B');
```



Error Bar Plots

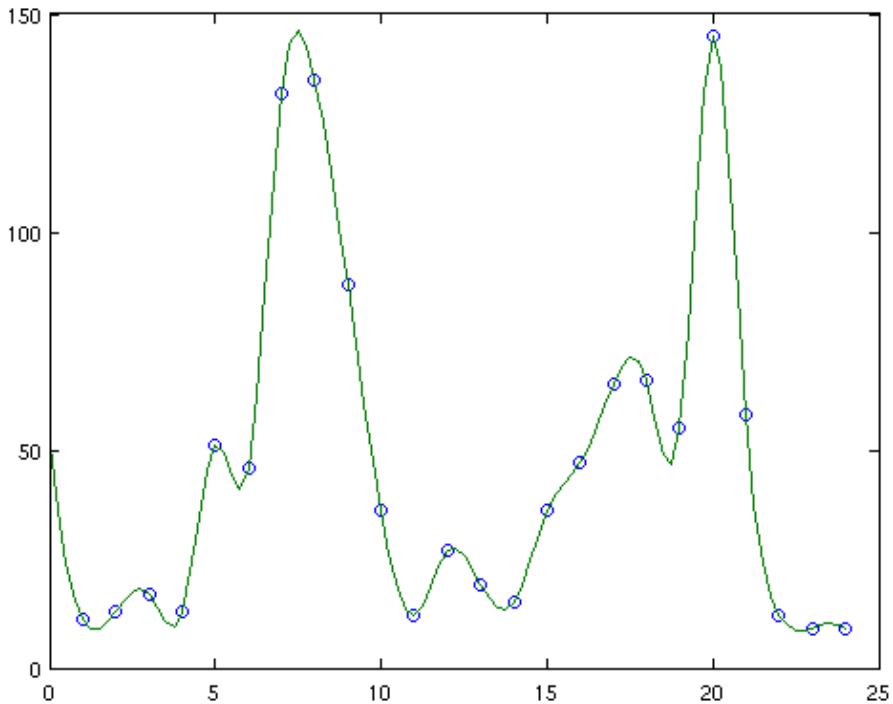
```
load count.dat;
y = mean(count,2);
e = std(count,1,2);
figure
errorbar(y,e,'xr')
```



Splines

Add interpolation

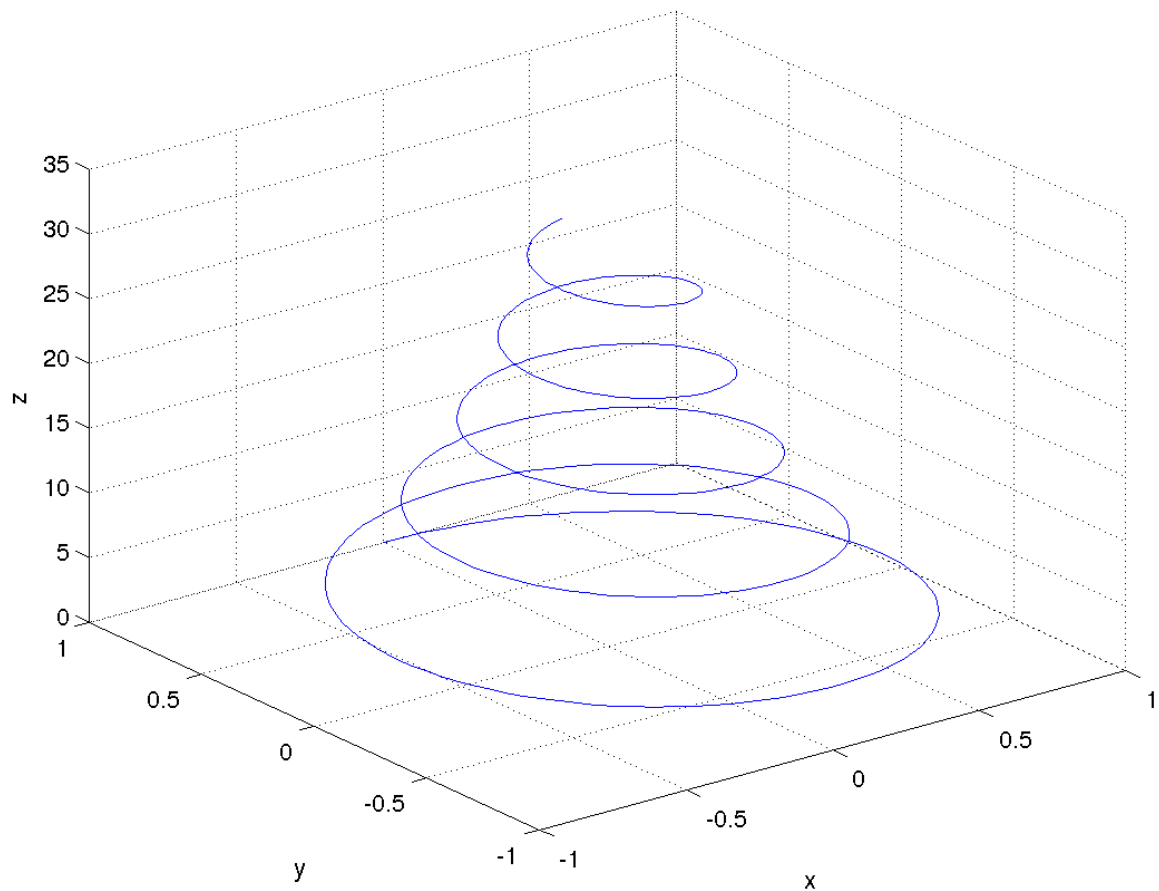
```
x=1:24
y=count(:,2)
xx=0:.25:24
yy=spline(x,y,xx)
plot(x,y,'o',xx,yy)
```



Three-Dimensional Line Plots

Plot in 3D the curve: $x = e^{-0.05t}\sin(t), y = e^{-0.05t}\cos(t), z = t$

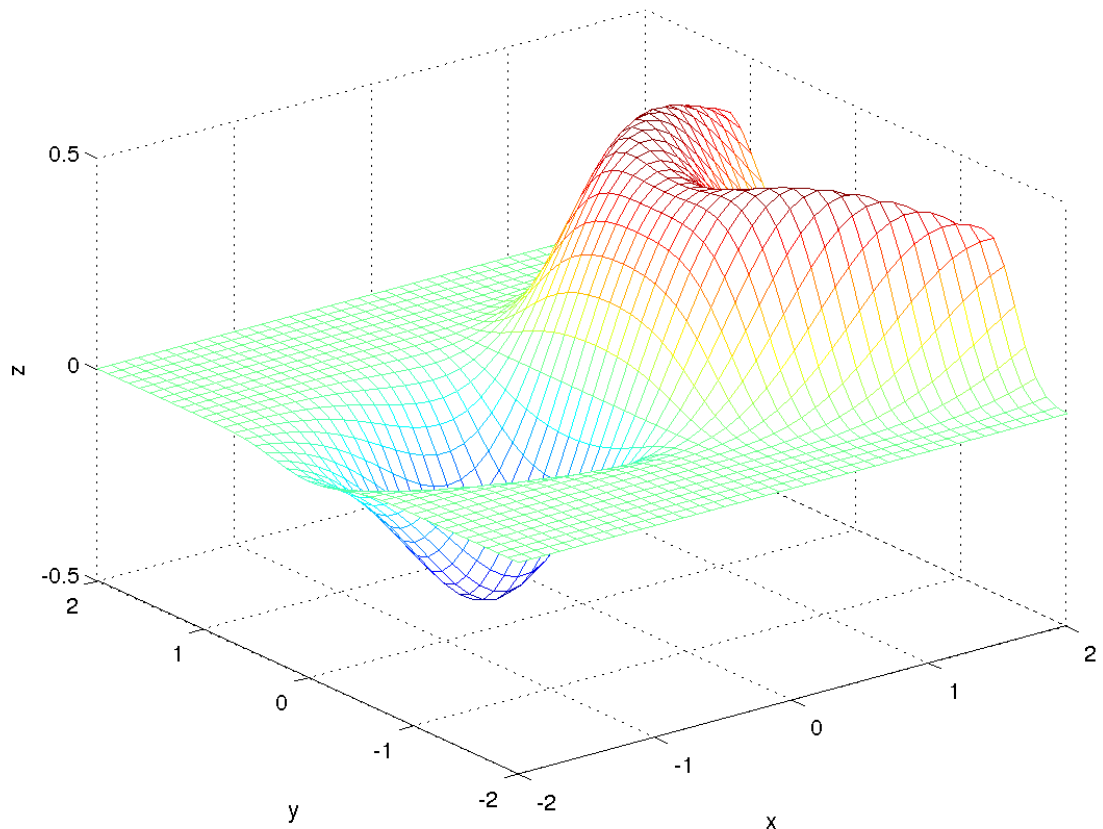
```
t = 0:pi/50:10*pi;
plot3(exp(-0.05*t).*sin(t), exp(-0.05*t).*cos(t), t)
xlabel('x'), ylabel('y'), zlabel('z'), grid
```



Surface Plots

Surface plot of the function $z = xe^{-[(x-y^2)^2+y^2]}$, for $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$ with a spacing of 0.1

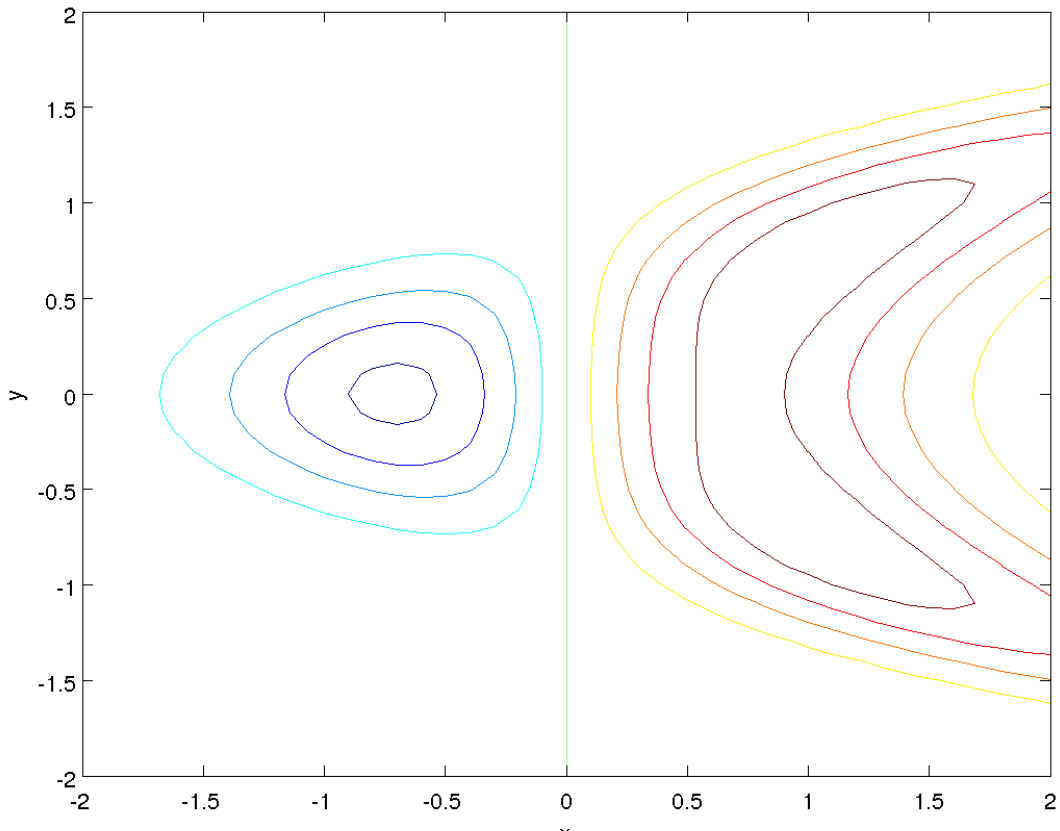
```
[X,Y] = meshgrid(-2:0.1:2);
Z = X.*exp(-((X-Y.^2).^2+Y.^2));
mesh(X,Y,Z), xlabel('x'), ylabel('y'), zlabel('z')
```



Contour Plots

Contour plot of the function $z = xe^{-[(x-y^2)^2+y^2]}$, for $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$ with a spacing of 0.1

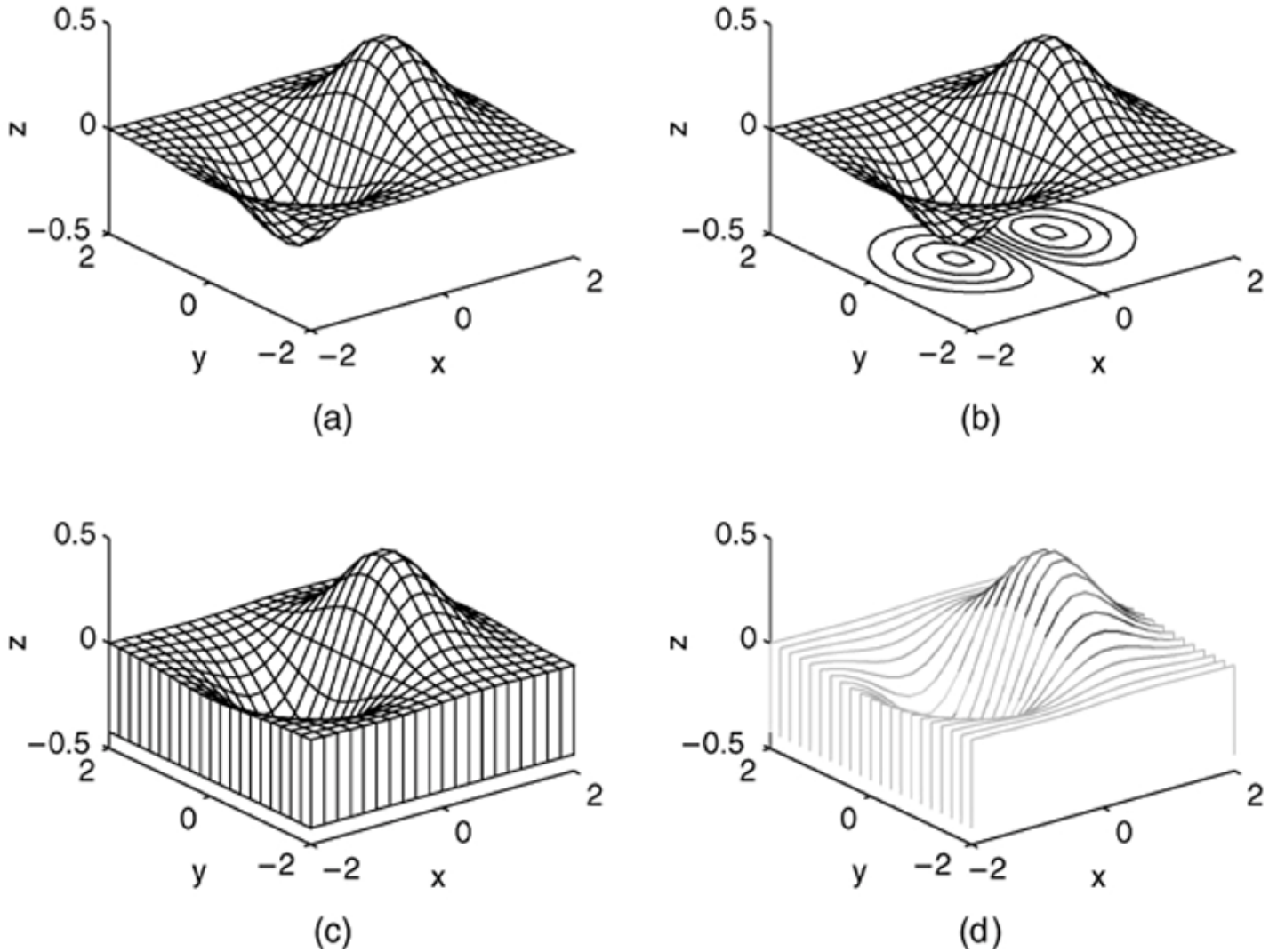
```
[X,Y] = meshgrid(-2:0.1:2);
Z = X.*exp(-((X-Y.^2).^2+Y.^2));
contour(X,Y,Z), xlabel('x'), ylabel('y')
```



Three-Dimensional Plotting Functions

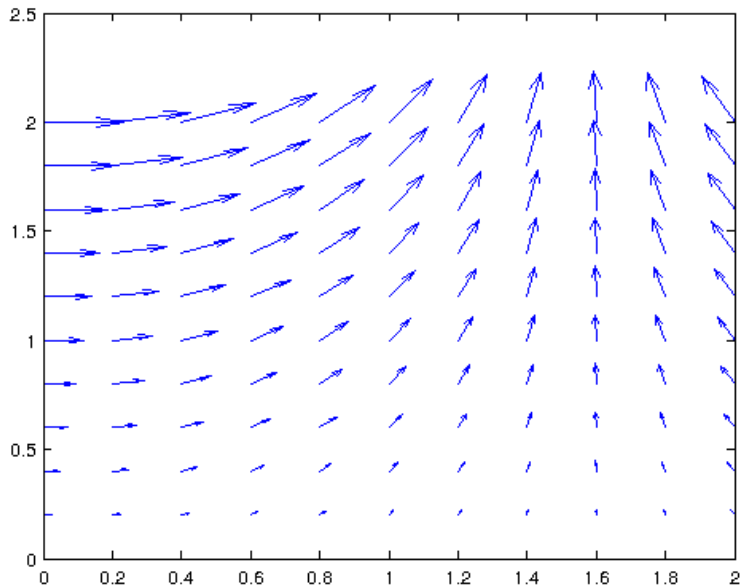
Function	Description
`contour(x,y,z)`	Creates a contour plot.
`mesh(x,y,z)`	Creates a 3D mesh surface plot.
`meshc(x,y,z)`	Same as mesh but draws contours under the surface.
`meshz(x,y,z)`	Same as mesh but draws vertical reference lines under the surface.
`surf(x,y,z)`	Creates a shaded 3D mesh surface plot.
`surfc(x,y,z)`	Same as surf but draws contours under the surface.
`[X,Y] = meshgrid(x,y)`	Creates the matrices X and Y from the vectors x and y to define a rectangular grid.
`[X,Y] = meshgrid(x)`	Same as `[X,Y]= meshgrid(x,x)`.
`waterfall(x,y,z)`	Same as mesh but draws mesh lines in one direction only.

a) mesh, b) meshc, c) meshz, d) waterfall



Vector fields

Use `quiver` to display an arrow at each data point in x and y such that the arrow direction and length represent the corresponding values of the vectors u and v.




```
[x,y] = meshgrid(0:0.2:2,0:0.2:2);
u = cos(x).*y;
v = sin(x).*y;

figure
quiver(x,y,u,v)
```

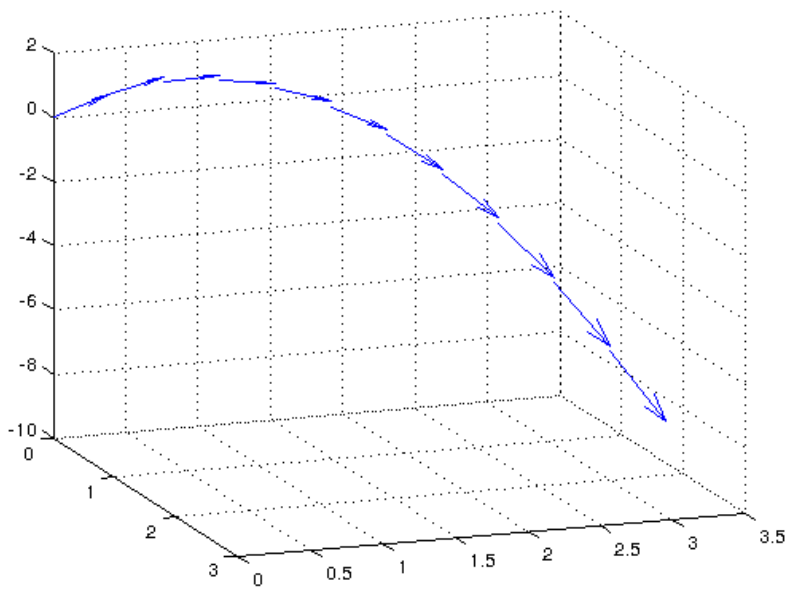
Vector fields

Projectile Path Over Time - `quiver3`

$$\mathbf{p}(t) = \mathbf{v}t + \frac{\mathbf{a}t^2}{2}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} t + \frac{1}{2} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} t^2$$

$$= \begin{bmatrix} 2 \\ 3 \\ 10 \end{bmatrix} t + \frac{1}{2} \begin{bmatrix} 0 \\ 0 \\ -32 \end{bmatrix} t^2$$




```
vz = 10; % velocity constant
a = -32; % acceleration constant
% Calculate z as the height as time varies
      from 0 to 1.
t = 0:.1:1;
z = vz*t + 1/2*a*t.^2;
% Calculate the position in the x-
      direction and y-direction.
vx = 2;
x = vx*t;
vy = 3;
y = vy*t;
% Compute the components of the velocity
      vectors and display the vectors
u = gradient(x);
v = gradient(y);
w = gradient(z);
scale = 0;

figure
quiver3(x,y,z,u,v,w,scale)
% Change the viewpoint of the axes to
      [70,18].
view([70,18])
```

Guidelines for Making Plots

- Should the experimental setup from the exploratory phase be redesigned to increase conciseness or accuracy?
- What parameters should be varied? What variables should be measured?
- How are parameters chosen that cannot be varied?
- Can tables be converted into curves, bar charts, scatter plots or any other useful graphics?
- Should tables be added in an appendix?
- Should a 3D-plot be replaced by collections of 2D-curves?
- Can we reduce the number of curves to be displayed?
- How many figures are needed?
- Should the x-axis be transformed to magnify interesting subranges?

- Should the x-axis have a logarithmic scale? If so, do the x-values used for measuring have the same basis as the tick marks?
- Make sure the each axis is labeled with the name of the quantity being plotted and its units.
- Make tick marks regularly paced and easy to interpret and interpolate, eg, 0.2, 0.4, rather than 0.23, 0.46
- Use the same scale limits and tick spacing on each plot if you need to compare information on more than one plot.
- Is the range of x-values adequate?
- Do we have measurements for the right x-values, i.e., nowhere too dense or too sparse?
- Should the y-axis be transformed to make the interesting part of the data more visible?
- Should the y-axis have a logarithmic scale?
- Is it misleading to start the y-range at the smallest measured value? (if not too much space wasted start from 0)
- Clip the range of y-values to exclude useless parts of curves?

- Can we use banking to 45^o?
- Are all curves sufficiently well separated?
- Can noise be reduced using more accurate measurements?
- Are error bars needed? If so, what should they indicate? Remember that measurement errors are usually not random variables.
- Connect points belonging to the same curve.
- Only use splines for connecting points if interpolation is sensible.
- Do not connect points belonging to unrelated owners.
- Use different point and line styles for different curves.
- Use the same styles for corresponding curves in different graphs.
- Place labels defining point and line styles in the right order and without concealing the curves.

- Captions should make figures self contained.
- Give enough information to make experiments reproducible.
- Golden ratio rule: make the graph wider than higher [Tufte 1983].
- Rule of 7: show at most 7 curves (omit those clearly irrelevant).
- Avoid: explaining axes, connecting unrelated points by lines, cryptic abbreviations, microscopic lettering, pie charts

Demos

Try!

```
demo 'matlab'
```

Summary

- Overview of MATLAB environment
- Overview of MATLAB programming and arrays
- Linear Algebra in MATLAB
  (Matrix and element-by-element operations)
- Solving linear systems in MATLAB
- Programming: Control structures
- Writing your own Functions
- Graphics: basic and advanced plotting
- Efficiency issues