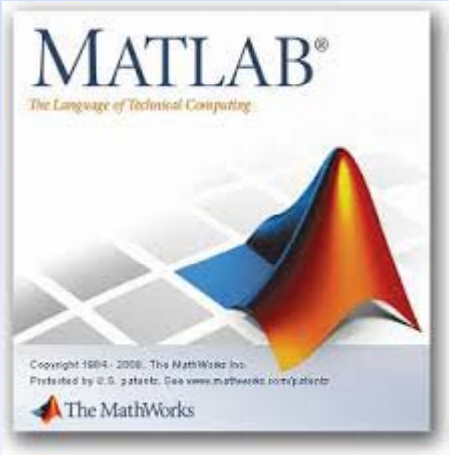
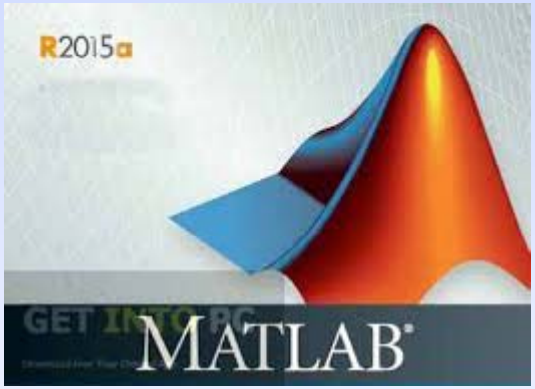
Introduction to MATLAB

part-1: basic operations





The MATLAB Environment

- MATLAB uses three primary windows-
  - Command window - used to enter commands and data
  - Graphics window(s) - used to display plots and graphics
  - Edit window - used to create and edit M-files (programs)

Calculator Mode

- The MATLAB command widow can be used as a calculator where you can type in commands line by line.
- Whenever a calculation is performed, MATLAB will assign the result to the built-in variable `ans`
- Example:

```
>> 55 - 16
ans =
     39
```

MATLAB Variables

- While using the `ans` variable may be useful for performing quick calculations, its transient nature makes it less useful for programming.
- MATLAB allows you to assign values to variable names. This results in the storage of values to memory locations corresponding to the variable name.
- MATLAB can store individual values as well as arrays; it can store numerical data and text (which is actually stored numerically as well).
- MATLAB does not require that you pre-initialize a variable; if it does not exist, MATLAB will create it for you.

Scalars

- To assign a single value to a variable, simply type the variable name, the = sign, and the value:

```
>> a = 4
a =
   4
```

- Note that variable names must start with a letter, though they can contain letters, numbers, and the underscore (_) symbol

Scalars (cont)

- You can tell MATLAB not to report the result of a calculation by appending the semi-colon (`;`) to the end of a line. The calculation is still performed.
- You can ask MATLAB to report the value stored in a variable by typing its name:

```
>> a
a =
   4
```

Scalars (cont)

- You can use the complex variable `i` (or `j`) to represent the unit imaginary number.
- You can tell MATLAB to report the values back using several different formats using the format command. Note that the values are still *stored* the same way, they are just displayed on the screen differently. Some examples are:
  - `short` - scaled fixed-point format with 5 digits
  - `long` - scaled fixed-point format with 15 digits for double and 7 digits for single
  - `short eng` - engineering format with at least 5 digits and a power that is a multiple of 3 (useful for SI prefixes)

Format Examples

```
>> format short; pi
ans =
    3.1416
>> format long; pi
ans =
   3.14159265358979
>> format short eng; pi
ans =
     3.1416e+000
>> pi*10000
ans =
    31.4159e+003
```

- **Note -** the format remains the same unless another `format` command is issued

Arrays, Vectors, and Matrices

- MATLAB can automatically handle rectangular arrays of data - one-dimensional arrays are called *vectors* and two-dimensional arrays are called *matrices*.
- Arrays are set off using square brackets [and] in MATLAB
- Entries within a row are separated by spaces or commas
- Rows are separated by semicolons

Array Examples

```
>> a = [1 2 3 4 5 ]
a =
     1     2     3     4     5
>> b = [2;4;6;8;10]
b =
     2
     4
     6
     8
    10
```

- Note 1 - MATLAB does not *display* the brackets
- Note 2 - if you are using a monospaced font, such as Courier, the displayed values should line up properly

Matrices

- A 2-D array, or matrix, of data is entered row by row, with spaces (or commas) separating entries within the row and semicolons separating the rows:

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

Useful Array Commands

- The transpose operator (apostrophe) can be used to flip an array over its own diagonal. For example, if `b` is a row vector, `b'` is a column vector containing the complex conjugate of `b`.
- The command window will allow you to separate rows by hitting the Enter key - script files and functions will allow you to put rows on new lines as well.
- The `who` command will report back used variable names; `whos` will also give you the size, memory, and data types for the arrays.

Accessing Array Entries

- Individual entries within a array can be both read and set using either the *index* of the location in the array or the row and column.
- The index value starts with 1 for the entry in the top left corner of an array and increases down a column - the following shows the indices for a 4 row, 3 column matrix:

```
1       5       9
2       6      10
3       7      11
4       8      12
```

Accessing Array Entries (cont)

- Assuming some matrix C:

```
C =
   2      4      9
   3      3     16
   3      0      8
  10     13     17
```

- `C(2)` would report 3
- `C(4)` would report 10
- `C(13)` would report an error!
- Entries can also be access using the row and column:
- `C(2,1)` would report 3
- `C(3,2)` would report 0
- `C(5,1)` would report an error!

Array Creation - Built In

- There are several built-in functions to create arrays:
  - `zeros(r,c)` will create an `r` row by `c` column matrix of zeros
  - `zeros(n)` will create an `n` by `n` matrix of zeros
  - `ones(r,c)` will create an `r` row by `c` column matrix of ones
  - `ones(n)` will create an `n` by `n` matrix one ones
- `help elmat` has, among other things, a list of the elementary matrices

Array Creation - Colon Operator

- The colon operator : is useful in several contexts. It can be used to create a linearly spaced array of points using the notation
  `start:diffval:limit`
  where `start` is the first value in the array, `diffval` is the difference between successive values in the array, and `limit` is the *boundary* for the last value (though not necessarily the last value).

```
>>1:0.6:3
ans =
    1.0000   1.6000   2.2000
2.8000
```

Colon Operator - Notes

- If `diffval` is omitted, the default value is 1:

```
>>3:6
ans =
     3     4     5     6
```

- To create a decreasing series, `diffval` must be negative:

```
>> 5:-1.2:2
ans =
    5.0000    3.8000    2.6000
```

- If `start+diffval>limit` for an increasing series or `start+diffval<limit` for a decreasing series, an empty matrix is returned:

```
>>5:2
ans =
   Empty matrix: 1-by-0
```

- To create a column, transpose the output of the colon operator, not the limit value; that is, `(3:6)'` not `3:6'`

Array Creation - `linspace`

- To create a row vector with a specific number of linearly spaced points between two numbers, use the `linspace` command.
- `linspace(x1, x2, n)` will create a linearly spaced array of `n` points between `x1` and `x2`

```
>>linspace(0, 1, 6)
ans =
   0   0.2000   0.4000   0.6000   0.8000
1.0000
```

- If `n` is omitted, 100 points are created.
- To generate a column, transpose the output of the `linspace` command.

Array Creation - `logspace`

- To create a row vector with a specific number of logarithmically spaced points between two numbers, use the `logspace` command.
- `logspace(x1, x2, n)` will create a logarithmically spaced array of `n` points between 10^{x1} and 10^{x2}

```
>>logspace(-1, 2, 4)
ans =
   0.1000    1.0000   10.0000  100.0000
```

- If `n` is omitted, 50 points are created.
- To generate a column, transpose the output of the `logspace` command.

Character Strings & Ellipsis

- Alphanumeric constants are enclosed by apostrophes (')

```
>> f = ‘My name is ';
>> s = ‘Bond'
```

- Concatenation: pasting together of strings

```
>> x = [f s]
x =
My name is Bond
```

- Ellipsis (...): Used to continue long lines

```
>> a = [1 2 3 4 5 ...
 6 7 8]
 a =
      1     2     3     4     5     6     7     8
```

- You cannot use an ellipsis within single quotes to continue a string. But you can piece together shorter strings with ellipsis

```
>> quote = ['Any fool can make a rule,' ...
 ' and any fool will mind it']
 quote =
 Any fool can make a rule, and any fool will mind it
```

Mathematical Operations

- Mathematical operations in MATLAB can be performed on both scalars and arrays.
- The common operators, in order of priority, are:

^	Exponentiation	4^2 = 16
-	Negation (unary operation)	-8 = -8
* /	Multiplication and Division	2*pi = 6.2832 pi/4 = 0.7854
\	Left Division	6\2 = 0.3333
+ -	Addition and Subtraction	3+5 = 8 3-5 = -2

Order of Operations

- The order of operations is set first by parentheses, then by the default order given above:
  - `y = -4 ^ 2` gives `y = -16`
    since the exponentiation happens first due to its higher default priority, but
  - `y = (-4) ^ 2` gives `y = 16`
    since the negation operation on the 4 takes place first

Complex Numbers

- All the operations above can be used with complex quantities (i.e. values containing an imaginary part entered using `i` or `j` and displayed using `i`)

```
>> x = 2+i*4; (or 2+4i, or 2+j*4, or 2+4j)
>> y = 16;
>> 3 * x
ans =
   6.0000 +12.0000i
>> x+y
ans =
  18.0000 + 4.0000i
>> x'
ans =
    2.0000 - 4.0000i
```

Vector-Matrix Calculations

- MATLAB can also perform operations on vectors and matrices.
- The * operator for matrices is defined as the *outer product* or what is commonly called "matrix multiplication."
  - The number of columns of the first matrix must match the number of rows in the second matrix.
  - The size of the result will have as many rows as the first matrix and as many columns as the second matrix.
  - The exception to this is multiplication by a 1×1 matrix, which is actually an array operation.
- The ^ operator for matrices results in the matrix being matrix-multiplied by itself a specified number of times.
  - Note - in this case, the matrix must be square!

Element-by-Element Calculations

- At times, you will want to carry out calculations item by item in a matrix or vector. The MATLAB manual calls these *array operations*. They are also often referred to as *element-by-element* operations.
- MATLAB defines **.*** and **./** (note the dots) as the array multiplication and array division operators.
  - For array operations, both matrices must be the same size *or* one of the matrices must be 1×1
- Array exponentiation (raising each element to a corresponding power in another matrix) is performed with **.^**
  - Again, for array operations, both matrices must be the same size *or* one of the matrices must be 1×1

Built-In Functions

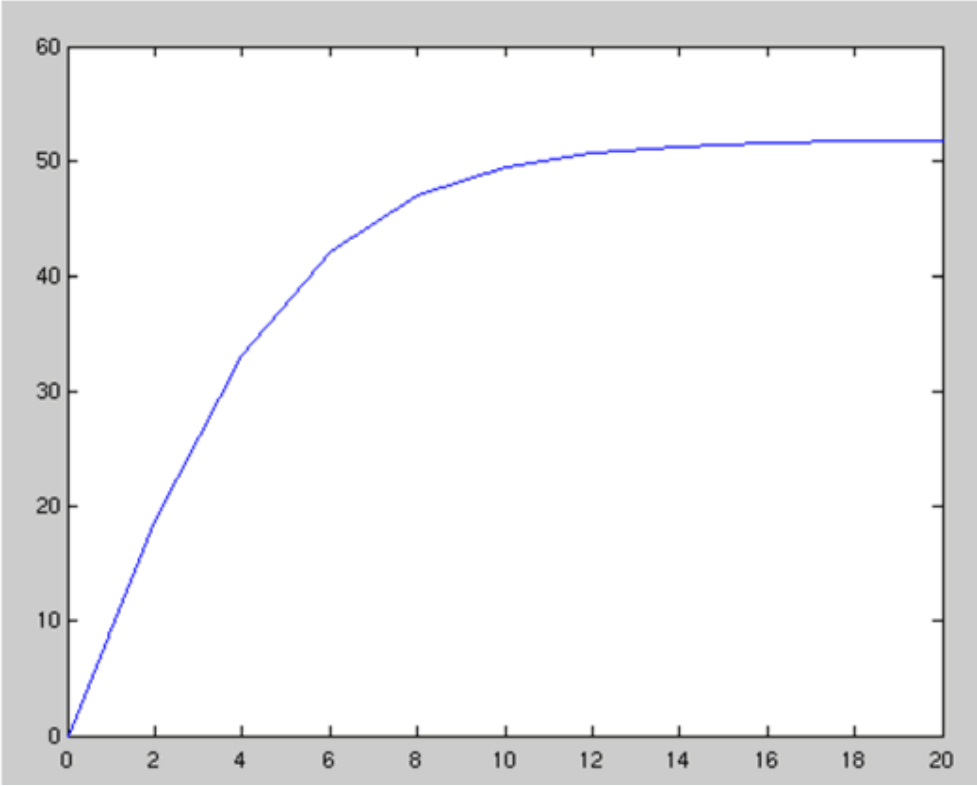
- There are several built-in functions you can use to create and manipulate data.
- The built-in help function can give you information about both what exists and how those functions are used:
  - `help elmat` will list the elementary matrix creation and manipulation functions, including functions to get information about matrices.
  - `help elfun` will list the elementary math functions, including trig, exponential, complex, rounding, and remainder functions.
- The built-in `lookfor` command will search help files for occurrences of text and can be useful if you know a function's purpose but not its name

Graphics

- MATLAB has a powerful suite of built-in graphics functions.
- Two of the primary functions are `plot` (for plotting 2-D data) and `plot3` (for plotting 3-D data).
- In addition to the plotting commands, MATLAB allows you to label and annotate your graphs using the `title`, `xlabel`, `ylabel`, and `legend` commands.

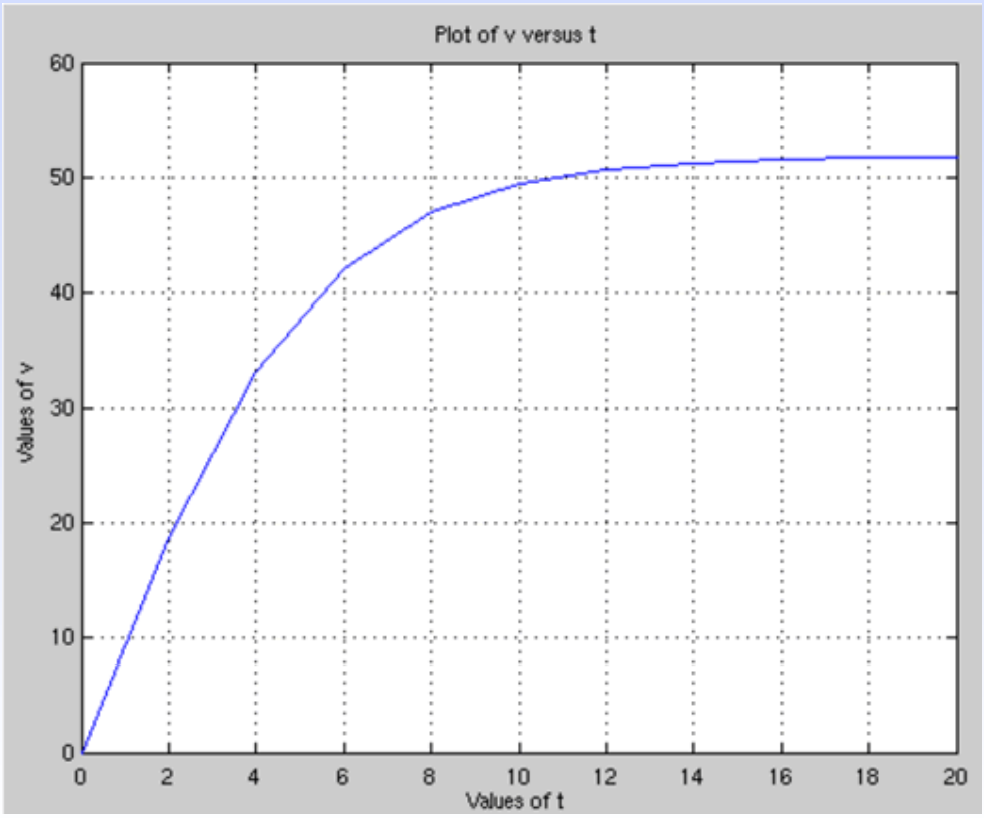
Plotting Example

```
t = [0:2:20]';
g = 9.81; m = 68.1; cd = 0.25;
v = sqrt(g*m/cd) * tanh(sqrt(g*cd/m)*t);
plot(t, v)
```



Plotting Annotation Example

```
title('Plot of v versus t')
xlabel('Values of t')
ylabel('Values of v')
grid
```



Plotting Options

- When plotting data, MATLAB can use several different colors, point styles, and line styles. These are specified at the end of the `plot` command using *plot specifiers*. More details: `>>help LineSpec`
- The default case for a single data set is to create a blue line with no points. If a line style is specified with no point style, no point will be drawn at the individual points; similarly, if a point style is specified with no point style, no line will be drawn.
- Examples of plot specifiers:
  - ‘ro:’ - red dotted line with circles at the points
  - ‘gd’ - green diamonds at the points with no line
  - ‘m--’ - magenta dashed line with no point symbols

Other Plotting Commands

- `hold on` and `hold off`
  - `hold on` tells MATLAB to keep the current data plotted and add the results of any further plot commands to the graph. This continues until the `hold off` command, which tells MATLAB to clear the graph and start over if another plotting command is given. `hold on` should be used *after* the first plot in a series is made.
- `subplot(m, n, p)`
  - `subplot` splits the figure window into an `mxn` array of small axes and makes the `p`th one active. Note - the first subplot is at the top left, then the numbering continues across the row. This is different from how elements are numbered within a matrix!

Some home practice exercises for students:

Problem-1:

2.4 The following matrix is entered in MATLAB:

```
>> A=[3 2 1;0:0.5:1;linspace(6, 8, 3)]
```

(a) Write out the resulting matrix.

(b) Use colon notation to write a single-line MATLAB command to multiply the second row by the third column and assign the result to the variable `C`.

Solution is given in the next slide

2.4

(a)

```
>> A=[3 2 1;0:0.5:1;linspace(6, 8, 3)]

A =
    3.0000    2.0000    1.0000
         0    0.5000    1.0000
    6.0000    7.0000    8.0000
```

(b)

```
>> C=A(2,:)*A(:,3)

C =
    8.5
```

Some home practice exercises for students:

Problem-2:

2.5 The following equation can be used to compute values of y as a function of x:

$$y = be^{-ax}\sin(bx)(0.012x^4 - 0.15x^3 + 0.075x^2 + 2.5x)$$

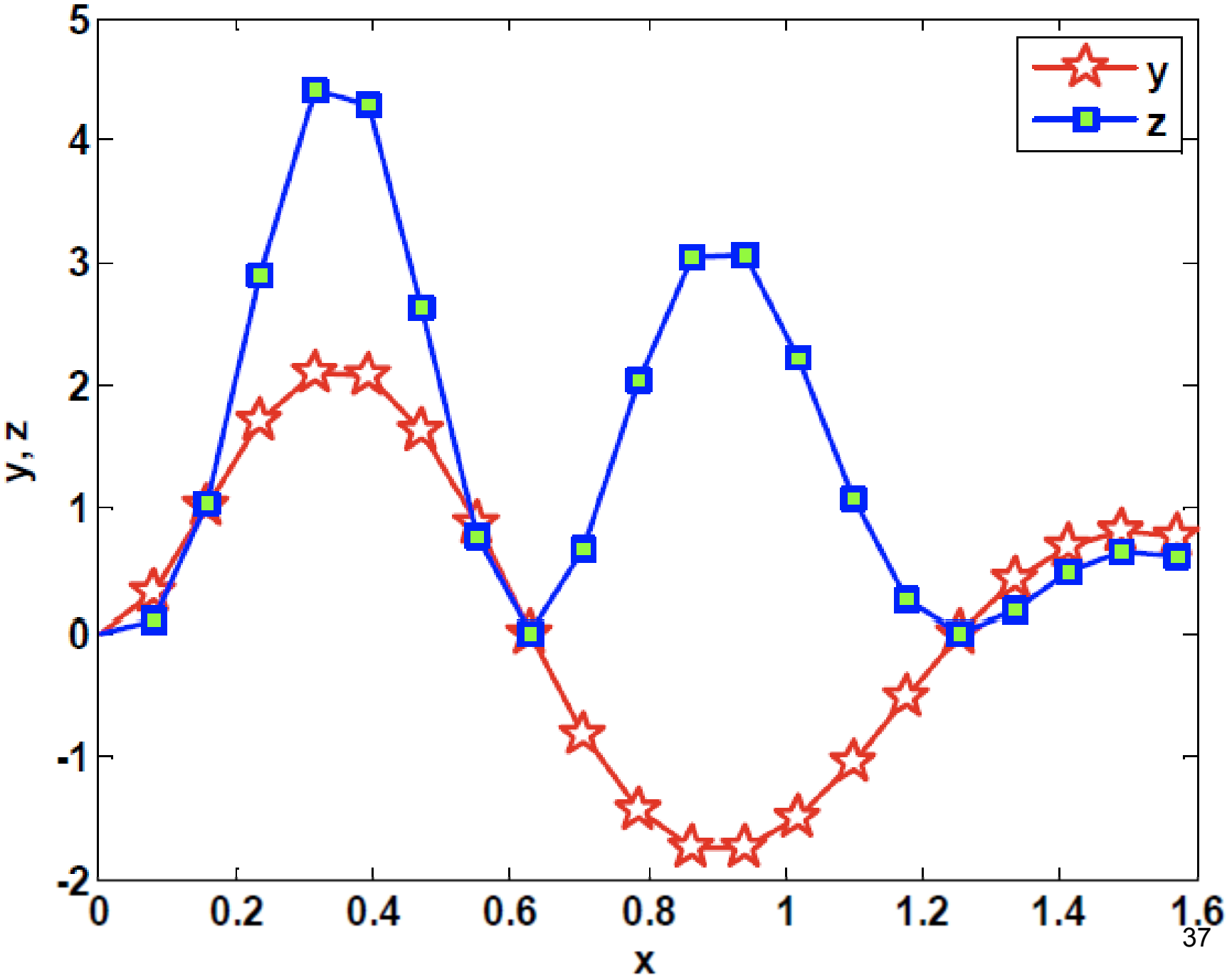
where a and b are parameters. Write the equation for implementation with MATLAB, where $a = 2$, $b = 5$, and x is a vector holding values from 0 to $\pi/2$ in increments of $\Delta x = \pi/40$. Employ the minimum number of periods (i.e., dot notation) so that your formulation yields a vector for y. In addition, compute the vector $z = y^2$ where each element holds the square of each element of y. Combine x, y, and z into a matrix w, where each column holds one of the variables, and display w using the `short g` format. In addition, generate a labeled plot of y and z versus x. Include a legend on the plot (use `help` to understand how to do this). For y, use a 1.5-point, dashdotted red line with 14-point, red-edged, white-faced pentagram-shaped markers. For z, use a standard-sized (i.e., default) solid blue line with standard-sized, blue-edged, green-faced square markers.

Solution is given in the next slide

2.5

```
format short g
a=2;b=5;
x=0:pi/40:pi/2;
y=b*exp(-a*x).*sin(b*x).*(0.012*x.^4-0.15*x.^3+0.075*x.^2+2.5*x);
z=y.^2;
w = [x' y' z']
plot(x,y,'-.pr','LineWidth',1.5,'MarkerSize',14,...
'MarkerEdgeColor','r','MarkerFaceColor','w')
hold on
plot(x,z,'-sb','MarkerFaceColor','g')
xlabel('x'); ylabel('y, z'); legend('y','z')
hold off
```

```
w =
         0              0              0
    0.07854        0.32172        0.10351
    0.15708         1.0174         1.0351
    0.23562          1.705         2.9071
    0.31416         2.1027         4.4212
     0.3927         2.0735         4.2996
    0.47124         1.6252         2.6411
    0.54978        0.87506        0.76573
    0.62832  2.7275e-016  7.4392e-032
    0.70686       -0.81663        0.66689
     0.7854         -1.427         2.0365
    0.86394        -1.7446         3.0437
    0.94248        -1.7512         3.0667
      1.021        -1.4891         2.2173
     1.0996        -1.0421         1.0859
     1.1781       -0.51272        0.26288
     1.2566 -2.9683e-016   8.811e-032
     1.3352        0.41762         0.1744
     1.4137        0.69202         0.4789
     1.4923        0.80787        0.65265
     1.5708        0.77866        0.60631
```


y
z
y, z
x
5
4
3
2
1
0
-1
-2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6

Some home practice exercises for students:

Problem-3:

2.7 The standard normal probability density function is a bell-shaped curve that can be represented as

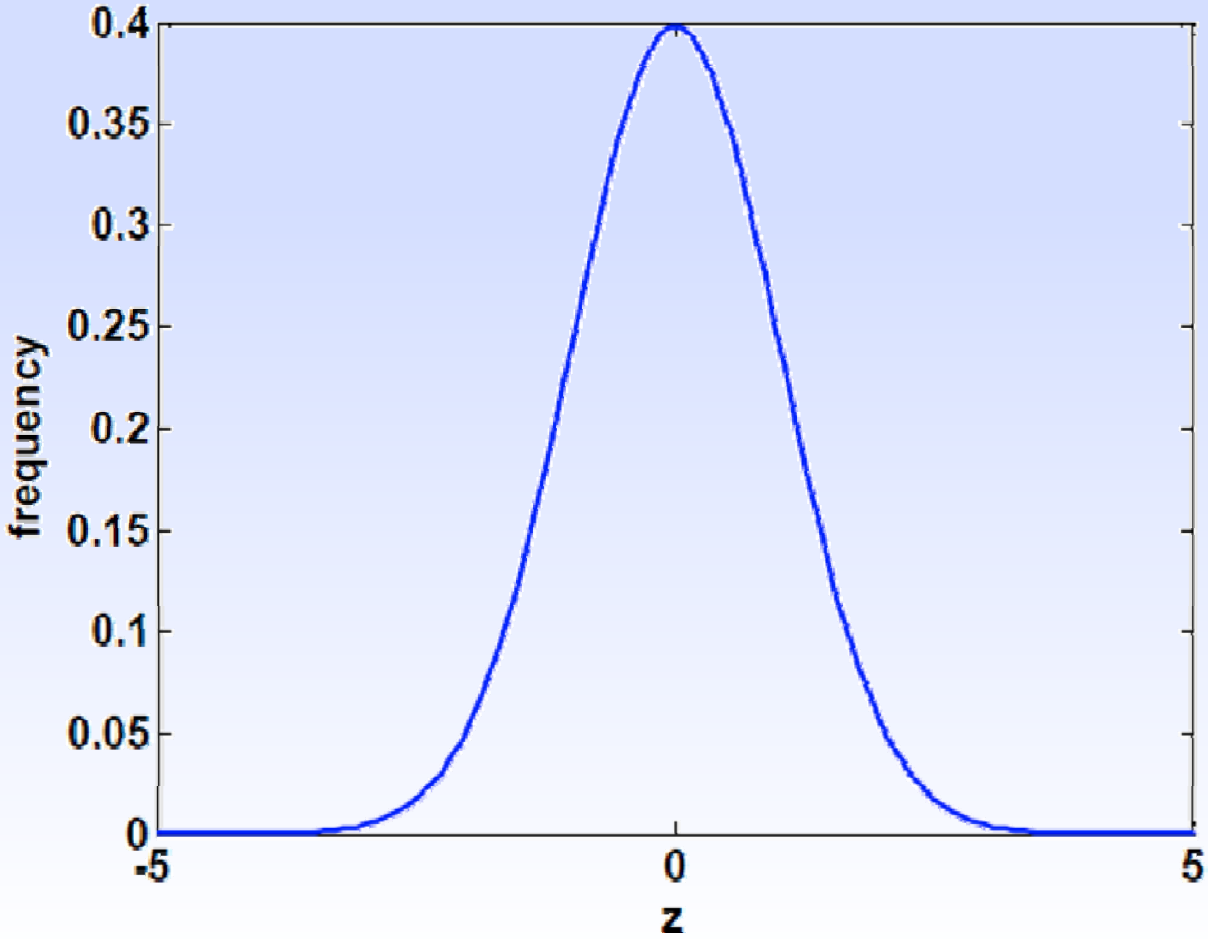
$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$$

Use MATLAB to generate a plot of this function from $z = -5$ to 5. Label the ordinate as frequency and the abscissa as z.

Solution is given in the next slide

2.7

```
>> z = linspace(-4,4);
>> f = 1/sqrt(2*pi)*exp(-z.^2/2);
>> plot(z,f)
>> xlabel('z')
>> ylabel('frequency')
```



Some home practice exercises for students:

Problem-4:

2.13 Here are some wind tunnel data for force (F) versus velocity (v):

v, m/s	10	20	30	40	50	60	70	80
F, N	25	70	380	550	610	1220	830	1450

These data can be described by the following function:

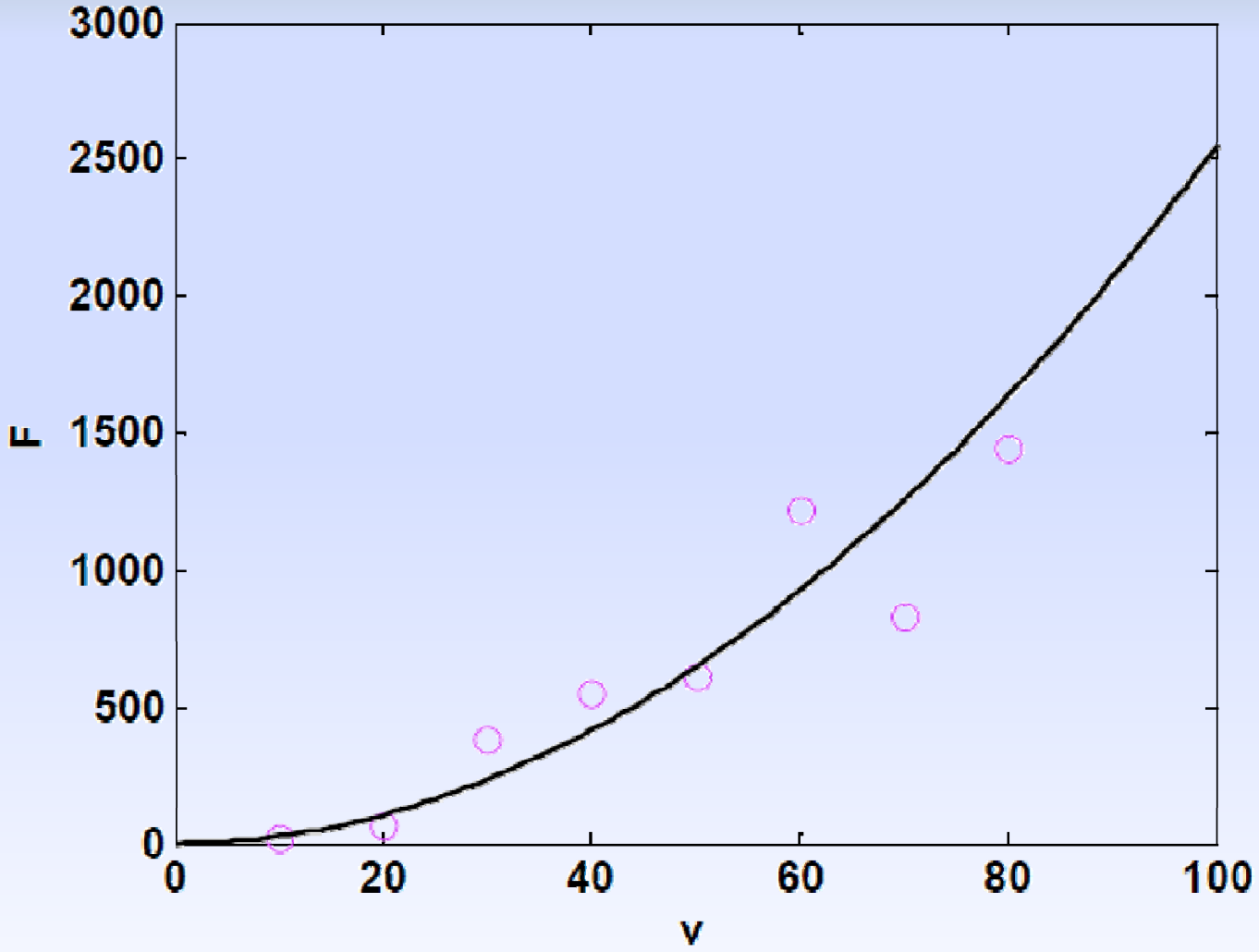
$$F = 0.2741v^{1.9842}$$

Use MATLAB to create a plot displaying both the data (using circular magenta symbols) and the function (using a black dash-dotted line). Plot the function for $v = 0$ to 100 m/s and label the plot's axes.

Solution is given in the next slide

2.13

```
>> v = 10:10:80;
>> F = [25 70 380 550 610 1220 830 1450];
>> vf = 0:100;
>> Ff = 0.2741*vf.^1.9842;
>> plot(v,F,'om',vf,Ff,'-.k')
>> xlabel('v');ylabel('F');
```


3000
2500
2000
1500
1000
500
0
F
0
20
40
60
80
100
v

Some home practice exercises for students:

Problem-5:

2.15 The Maclaurin series expansion for the cosine is

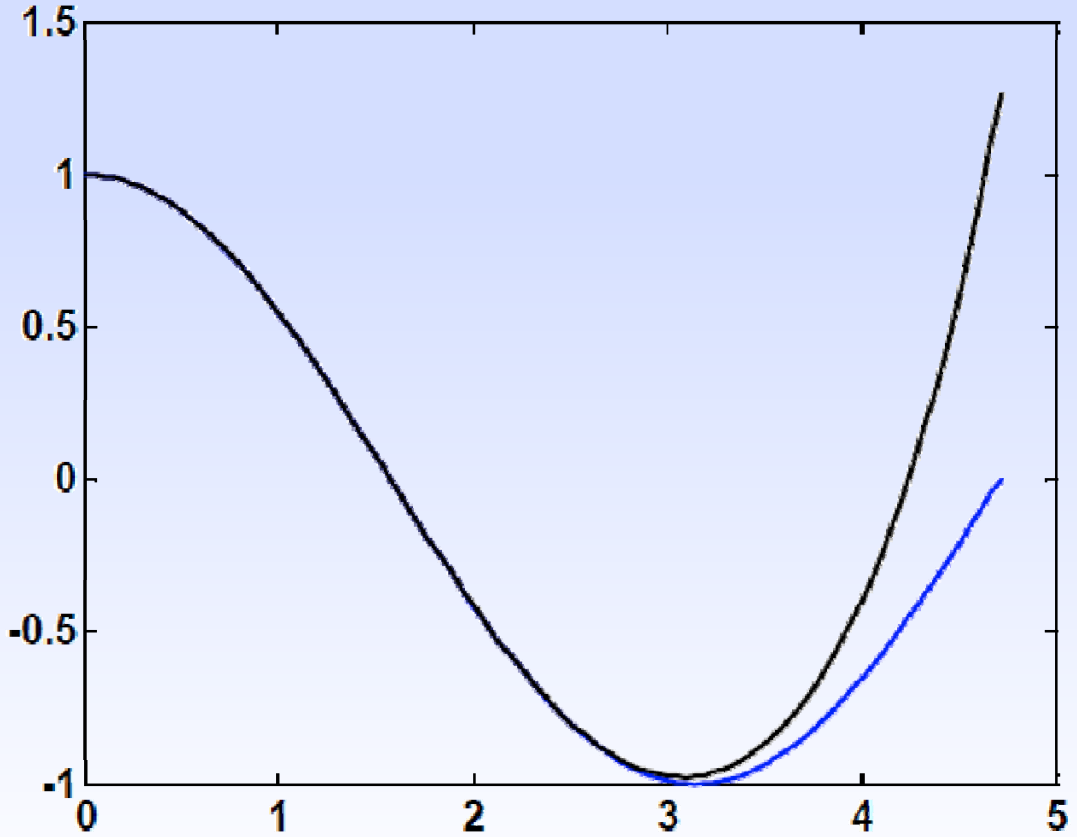
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

Use MATLAB to create a plot of the sine (solid line) along with a plot of the series expansion (black dashed line) up to and including the term $x^8/8!$. Use the built-in function `factorial` in computing the series expansion. Make the range of the abscissa from $x = 0$ to $3\pi/2$.

Solution is given in the next slide

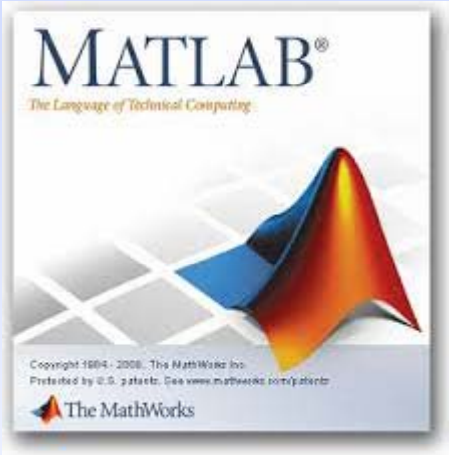
2.15

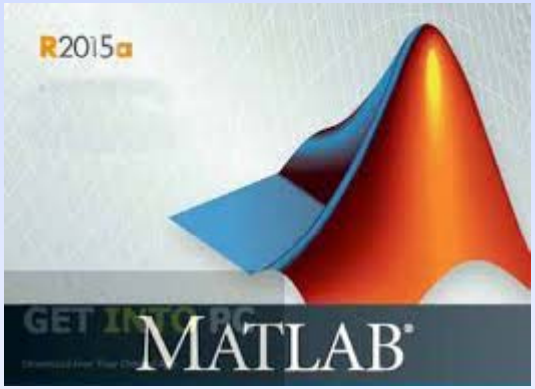
```
>> x = linspace(0,3*pi/2);
>> c = cos(x);
>> cf = 1-x.^2/2+x.^4/factorial(4)-
x.^6/factorial(6)+x.^8/factorial(8);
>> plot(x,c,x,cf,'k--')
```



Introduction to MATLAB

part-2: functions, scripts





M-files

- While commands can be entered directly to the command window, MATLAB also allows you to put commands in text files called *M-files*. *M-files* are so named because the files are stored with a `.m` extension.
- There are two main kinds of M-file
  - Script files
  - Function files

Script Files

- A *script file* is merely a set of MATLAB commands that are saved on a file - when MATLAB runs a script file, it is as if you typed the characters stored in the file on the command window.
- Scripts can be executed either by typing their name (without the .m) in the command window, by selecting the `Debug, Run` (or `Save and Run`) command in the editing window, or by hitting the `F5` key while in the editing window. Note that the latter two options will save any edits you have made, while the former will run the file as it exists on the drive.

Function Files

- *Function files* serve an entirely different purpose from script files. Function files can accept input arguments from and return outputs to the command window, but variables created and manipulated within the function do not impact the command window.

Function File Syntax

- The general syntax for a function is:

```
function outvar = funcname(arglist)
% helpcomments
statements
outvar = value;
```

where

- *outvar*: output variable name
- *funcname:* function's name
- *arglist:* input argument list; comma-delimited list of what the function calls values passed to it
- *helpcomments:* text to show with *help funcname*
- *statements:* MATLAB commands for the function

Subfunctions

- A function file can contain a single function, but it can also contain a *primary function* and one or more *subfunctions*
- The primary function is whatever function is listed first in the M-file - its function name should be the same as the file name.
- Subfunctions are listed below the primary function. Note that they are *only* accessible by the main function and subfunctions within the same M-file and *not* by the command window or any other functions or scripts.

Input

- The easiest way to get a value from the user is the input command:
  - `result = input(prompt)`
    displays the prompt string on the screen, waits for input from the keyboard, evaluates any expressions in the input, and returns the result.
  - `str = input(prompt,'s')`
    returns the entered text as a MATLAB string, without evaluating expressions.

Output

- The easiest way to display the value of a matrix is to type its name, but that will not work in function or script files. Instead, use the `disp` command

  ```
  disp(value)
  ```

  will show the *`value`* on the screen.
- If *`value`* is a string, enclose it in single quotes.

Formatted Output

- For formatted output, or for output generated by combining variable values with literal text, use the *fprintf* command:

```
fprintf('format', x, y,...)
```

  where *format* is a string specifying how you want the value of the variables *x*, *y*, and more to be displayed - including literal text to be printed along with the values.
- The values in the variables are formatted based on format codes.

Format and Control Codes

- Within the *format* string, the following format codes define how a numerical value is displayed:
  %d - integer format
  %e - scientific format with lowercase e
  %E - scientific format with uppercase E
  %f - decidmal format
  %g - the more compact of %e or %f
- The following control codes produce special results within the *format* string:
  \n - start a new line
  \t - tab
  \\ - print the \ character
- To print a ' put a pair of ' in the *format* string

Creating and Accessing Files

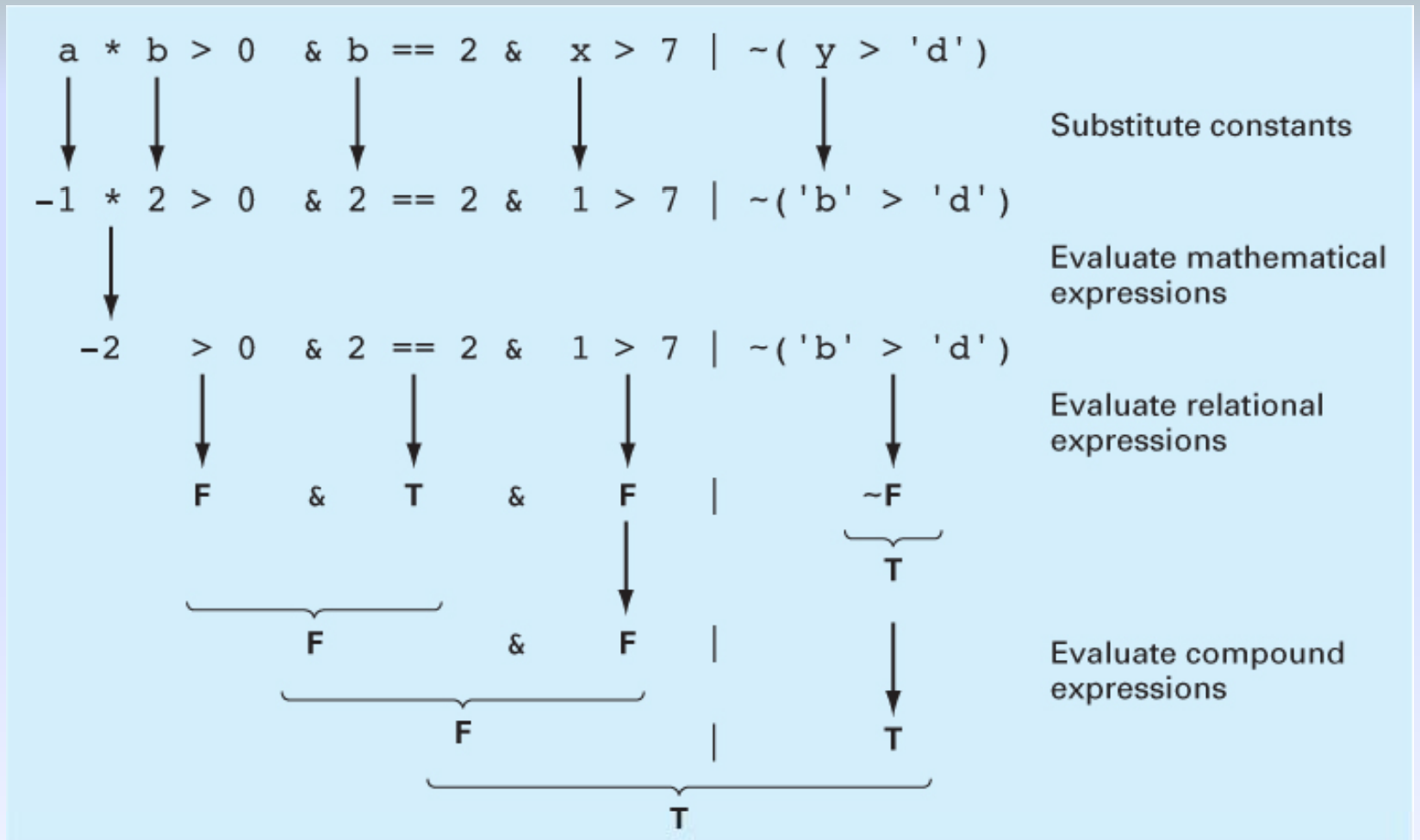
- MATLAB has a built-in file format that may be used to save and load the values in variables.
- `save` *`filename var1 var2 ... varn`* saves the listed variables into a file named *`filename`*`.mat`. If no variable is listed, all variables are saved.
- `load` *`filename var1 var2 ...varn`* loads the listed variables from a file named *`filename`*`.mat`. If no variable is listed, all variables in the file are loaded.
- Note - these are not text files!

ASCII Files

- To create user-readable files, append the flag `-ascii` to the end of a `save` command. This will save the data to a text file in the same way that `disp` sends the data to a screen.
- Note that in this case, MATLAB does *not* append anything to the file name so you may want to add an extension such as *.txt* or *.dat*.
- To load a rectangular array from a text file, simply use the *load* command and the file name. The data will be stored in a matrix with the same name as the file (but without any extension).

Structured Programming

- Structured programming allows MATLAB to make decisions or selections based on conditions of the program.
- Decisions in MATLAB are based on the result of logical and relational operations and are implemented with `if, if…else`, and `if…elseif` structures.
- Selections in MATLAB are based on comparisons with a test expression and are implemented with `switch` structures.


a * b > 0 & b == 2 & x > 7 | ~(y > 'd')
Substitute constants
-1 * 2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
Evaluate mathematical expressions
-2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
Evaluate relational expressions
F & T & F | ~F
T
F & F |
Evaluate compound expressions
F |
T
T

Relational Operators

- Summary of relational operators in MATLAB:

Example	Operator	Relationship
`x == 0`	`==`	Equal
`unit ~= 'm'`	`~=`	Not equal
`a < 0`	`<`	Less than
`s > t`	`>`	Greater than
`3.9 <= a/3`	`<=`	Less than or equal to
`r >= 0`	`>=`	Greater than or equal to

Logical Operators

- ~x (Not): true if x is false (or zero); false otherwise
- x & y (And): true if both x and y are true (or non-zero)
- x | y (Or): true if either x or y are true (or non-zero)

Order of Operations

- Priority can be set using parentheses. After that, Mathematical expressions are highest priority, followed by relational operators, followed by logical operators. All things being equal, expressions are performed from left to right.
- *Not* is the highest priority logical operator, followed by *And* and finally *Or*
- Generally, do not combine two relational operators! If `x=5`, `3<x<4` should be false (mathematically), but it is calculated as an expression in MATLAB as: `3<5<4`, which leads to `true<4` at which point `true` is converted to `1`, and `1<4` is true!
- Use `(3<x)&(x<4)` to properly evaluate.

Decisions

- Decisions are made in MATLAB using `if` structures, which may also include several `elseif` branches and possibly a catch-all `else` branch.
- Deciding which branch runs is based on the result of *conditions* which are either true or false.
  - If an `if` tree hits a *true* condition, that branch (and that branch only) runs, then the tree terminates.
  - If an `if` tree gets to an `else` statement without running any prior branch, that branch will run.
- **Note** - if the *condition* is a matrix, it is considered true if and only if all entries are true (or non-zero).

Selections

- Selections are made in MATLAB using switch structures, which may also include a catch-all otherwise choice.
- Deciding which branch runs is based on comparing the value in some test expression with values attached to different cases.
  - If the test expression matches the value attached to a case, that case's branch will run.
  - If no cases match and there is an otherwise statement, that branch will run.

Loops

- Another programming structure involves loops, where the same lines of code are run several times. There are two types of loop:
  - A **for** loop ends after a specified number of repetitions established by the number of columns given to an index variable.
  - A **while** loop ends on the basis of a logical condition.

`for` Loops

- One common way to use a `for`…`end` structure is:

```
for index = start:step:finish
   statements
end
```

where the *`index`* variable takes on successive values in the vector created using the **`:`** operator.

Vectorization

- Sometimes, it is more efficient to have MATLAB perform calculations on an entire array rather than processing an array element by element. This can be done through *vectorization*.

for loop	Vectorization

`i = 0;`
`for t = 0:0.02:50`
`  i = i + 1;`
`  y(i) = cos(t);`
`end`
	`t = 0:0.02:50;`
	`y = cos(t);`

`while` Loops

- A while loop is fundamentally different from a for loop since while loops can run an indeterminate number of times. The general syntax is

```
while condition
  statements
end
```

where the *`condition`* is a logical expression. If the *`condition`* is true, the *`statements`* will run and when that is finished, the loop will again check on the *`condition`*.

- Note - though the *`condition`* may become false as the *`statements`* are running, the only time it matters is after all the statements have run.

Early Termination

- Sometimes it will be useful to break out of a for or while loop early - this can be done using a break statement, generally in conjunction with an if structure.
- Example:

```
x = 24
while (1)
   x = x - 5
   if x < 0, break, end
end
```

will produce x values of 24, 19, 14, 9, 4, and -1, then stop.

Animation

- Two ways to animate plots in MATLAB:
  - Using looping with simple plotting functions
    - This approach merely replots the graph over and over again.
    - Important to use the `axis` command so that the plots scales are fixed.
  - Using special function: `getframe` and `movie`
    - This allows you to capture a sequence of plots (`getframe`) and then play them back (`movie`).

Example

- The (x, y) coordinates of a projectile can be generated as a function of time, t,with the following parametric equations

$$x = v_0 \cos(\theta_0 t)$$

$$y = v_0 \sin(\theta_0 t) - 0.5\, gt^2$$

where v_0 = initial velocity (m/s)

θ_0 = initial angle (radians)

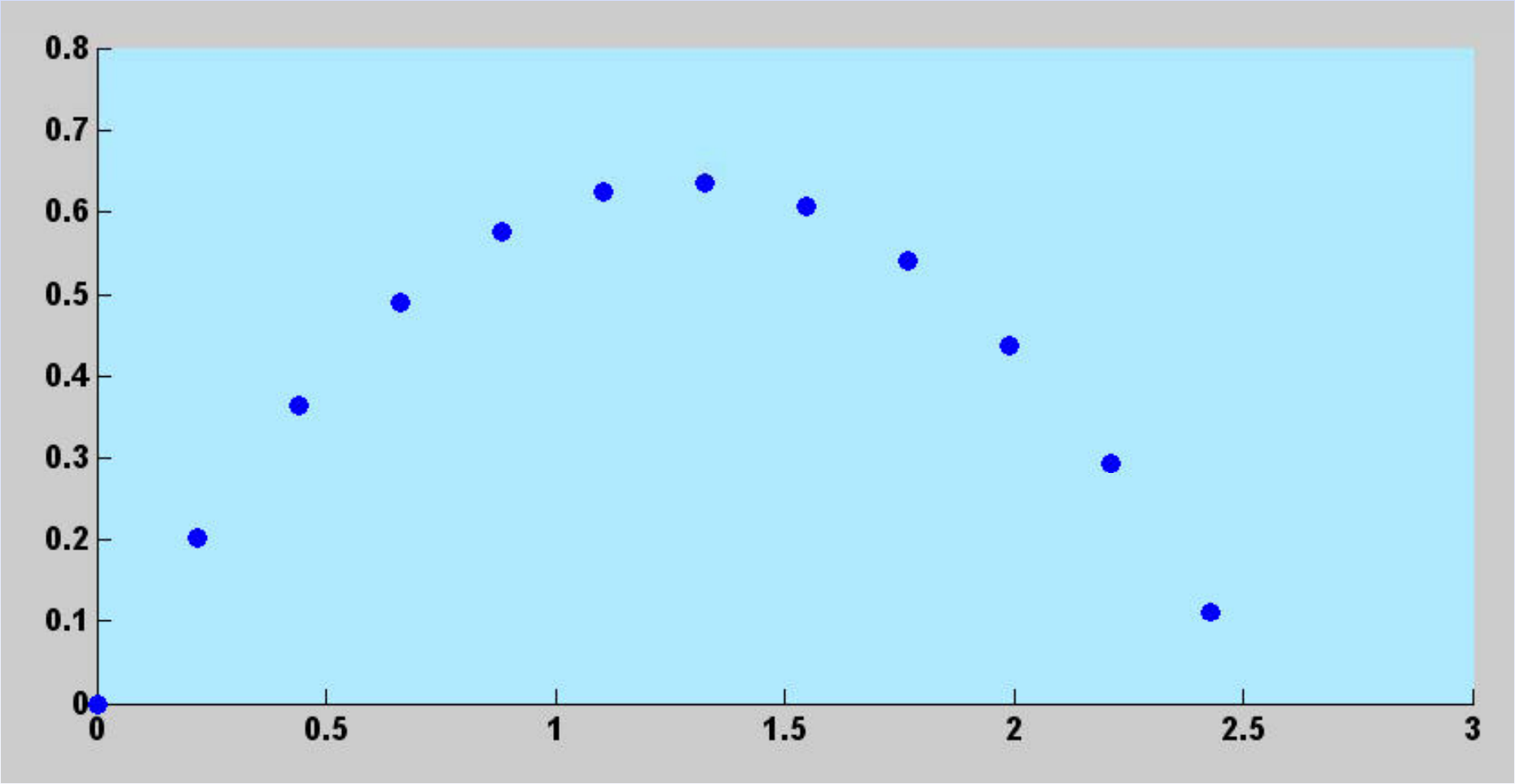
g = gravitational constant (= 9.81 m/s^2)

Script

- The following code illustrates both approaches:

```
clc,clf,clear
g=9.81; theta0=45*pi/180; v0=5;
t(1)=0;x=0;y=0;
plot(x,y,'o','MarkerFaceColor','b','MarkerSize',8)
axis([0 3 0 0.8])
M(1)=getframe;
dt=1/128;
for j = 2:1000
  t(j)=t(j-1)+dt;
  x=v0*cos(theta0)*t(j);
  y=v0*sin(theta0)*t(j)-0.5*g*t(j)^2;
  plot(x,y,'o','MarkerFaceColor','b','MarkerSize',8)
  axis([0 3 0 0.8])
  M(j)=getframe;
  if y<=0, break, end
end
pause
movie(M,1)
```

Result



Nesting and Indentation

- Structures can be placed within other structures. For example, the `statements` portion of a `for` loop can be comprised of an `if...elseif...else` structure.
- For clarity of reading, the `statements` of a structure are generally indented to show which lines of controlled are under the control of which structure.

Anonymous & Inline Functions

- *Anonymous functions* allow you to create a simple function without creating an M-file.
```
fhandle = @(arg1, arg2, ...) expression
```
- *Inline functions* are essentially the same as anonymous functions, but with a different syntax:
```
fhandle = inline('expression', 'arg1',
'arg2',...)
```
- Anonymous functions can access the values of variables in the workspace upon creation, while inlines cannot.

Function Functions

- *Function functions*: functions that operate on other functions passed as arguments.
- input argument: anonymous, inline function, the name of a built-in function, or the name of a M-file function.
- Allows more dynamic programming.
- *Example:*

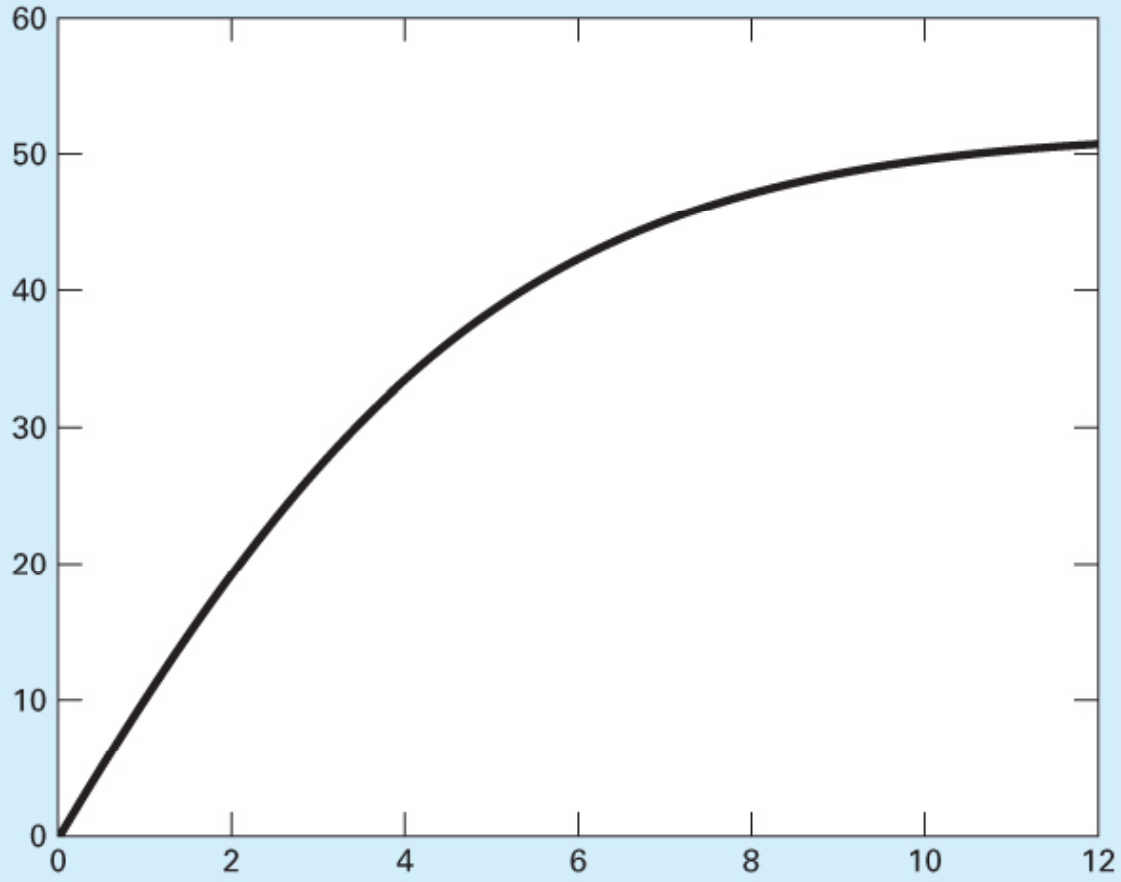
```
vel=@(t) sqrt(9.81*68.1/0.25)*tanh
(sqrt(9.81*0.25/68.1)* t);
```

Velocity of a bungee jumper with respect to time

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}}t\right)$$

To generate a plot from t = 0 to 12:

```
fplot(vel,[0 12])
```



A plot of velocity versus time generated with the `fplot` function.

- *Another simple way of plotting the same job is to use array operations with plot command.*
- *Example:*

```
t=0:0.01:12;
v=sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)
*t);
plot(t,v,'k','LineWidth',2)
```

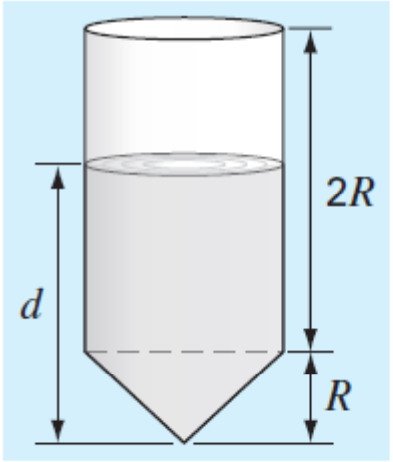
black

Line width equal to 2

Some home practice exercises for students:

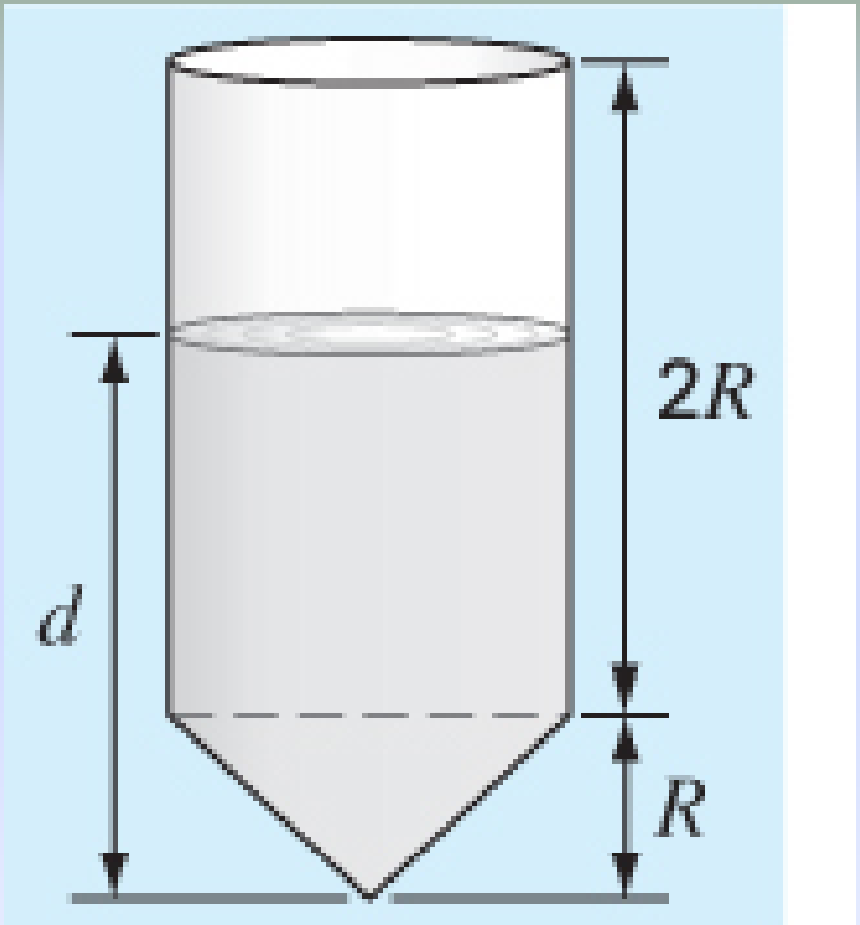
Problem-6: 3.1 Figure P3.1 shows a cylindrical tank with a conical base. If the liquid level is quite low, in the conical part, the volume is simply the conical volume of liquid. If the liquid level is midrange in the cylindrical part, the total volume of liquid includes the filled conical part and the partially filled cylindrical part.

Use decisional structures to write an M-file to compute the tank's volume as a function of given values of R and d. Design the function so that it returns the volume for all cases where the depth is less than $3R$. Return an error message ("Overtop") if you overtop the tank—that is, $d > 3R$. Test it with the following data:



R	0.9	1.5	1.3	1.3
d	1	1.25	3.8	4.0

Solution is given in the next slide



3.1 M-file

```
function Vol = tankvolume(R, d)
if d < R
    Vol = pi * d ^ 3 / 3;
elseif d <= 3 * R
    V1 = pi * R ^ 3 / 3;
    V2 = pi * R ^ 2 * (d - R);
    Vol = V1 + V2;
else
    Vol = 'overtop';
end
```

```
>> tankvolume(0.9,1)
>> tankvolume(1.5,1.25)
>> tankvolume(1.3,3.8)
>> tankvolume(1.3,4)
```

Some home practice exercises for students:

Problem-7:

3.6 Two distances are required to specify the location of a point relative to an origin in two-dimensional space (Fig. P3.6):

- The horizontal and vertical distances (x, y) in Cartesian coordinates.
- The radius and angle (r, θ) in polar coordinates.

It is relatively straightforward to compute Cartesian coordinates (x, y) on the basis of polar coordinates (r, θ). The reverse process is not so simple. The radius can be computed by the following formula:

$$r = \sqrt{x^2 + y^2}$$

If the coordinates lie within the first and fourth coordinates (i.e., $x > 0$), then a simple formula can be used to compute θ:

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

Some home practice exercises for students:

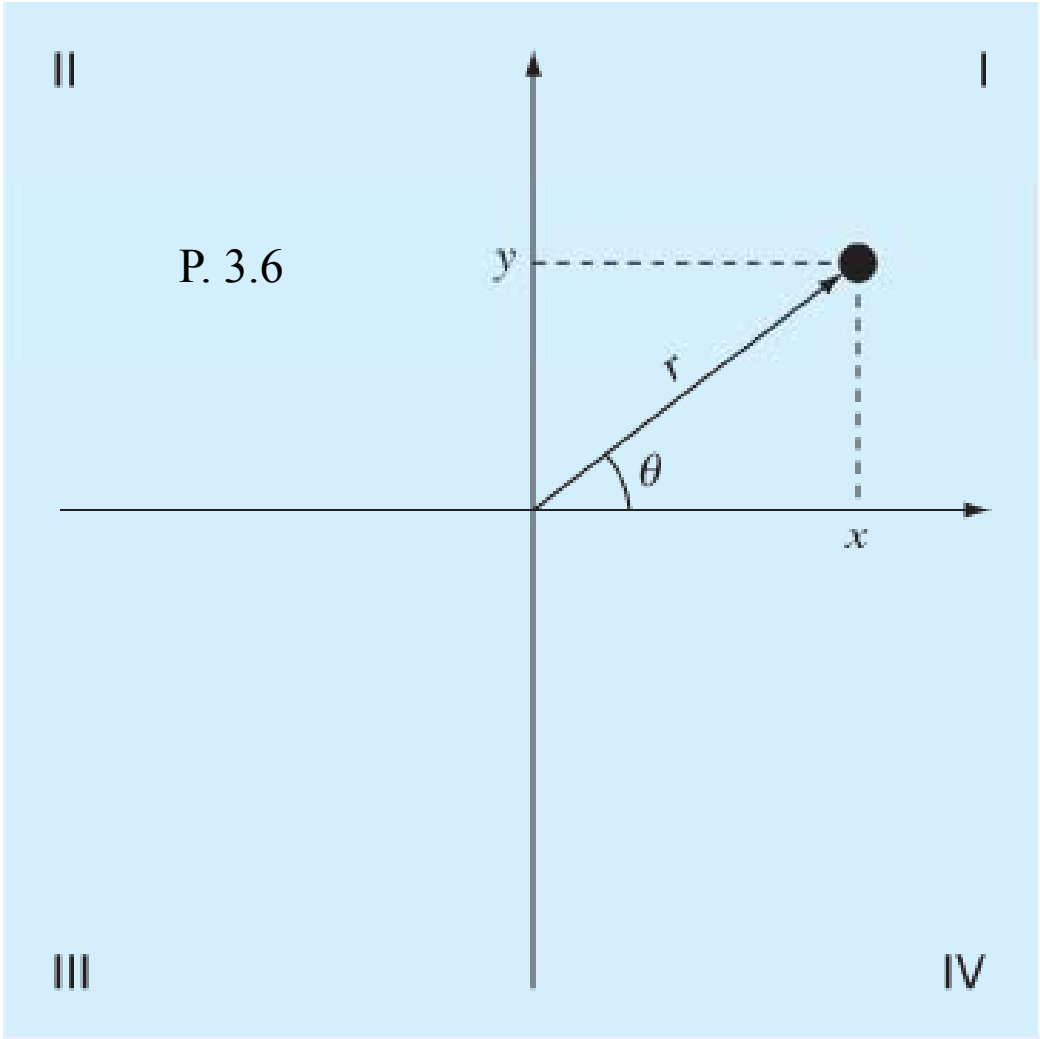
Problem-7:

The difficulty arises for the other cases. The following table summarizes the possibilities:

x	y	θ
<0	>0	$\tan^{-1}(y/x) + \pi$
<0	<0	$\tan^{-1}(y/x) - \pi$
<0	$=0$	π
$=0$	>0	$\pi/2$
$=0$	<0	$-\pi/2$
$=0$	$=0$	0

Write a well-structured M-file using `if...elseif` structures to calculate r and θ as a function of x and y. Express the final results for θ in degrees. Test your program by evaluating some values

Solution is given in the next slide




```
function [r, th] = polar(x, y)
r = sqrt(x .^ 2 + y .^ 2);
if x > 0
    th = atan(y/x);
elseif x < 0
    if y > 0
        th = atan(y / x) + pi;
    elseif y < 0
        th = atan(y / x) - pi;
    else
        th = pi;
    end
else
    if y > 0
        th = pi / 2;
    elseif y < 0
        th = -pi / 2;
    else
        th = 0;
    end
end
th = th * 180 / pi;
```

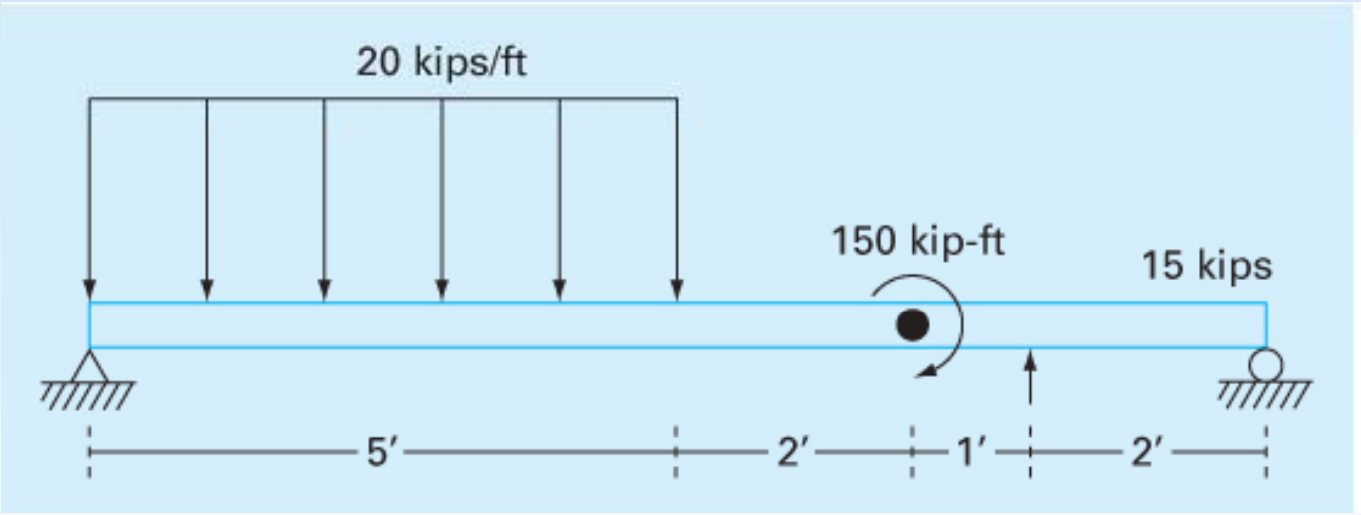
Example: [r,th]=polar(2,0)

Some home practice exercises for students:

Problem-8:

3.10 A simply supported beam is loaded as shown in Fig. P3.10. Using singularity functions, the displacement along the beam can be expressed by the equation:

$$u_y(x) = \frac{-5}{6}[\langle x-0\rangle^4 - \langle x-5\rangle^4] + \frac{15}{6}\langle x-8\rangle^3 + 75\langle x-7\rangle^2 + \frac{57}{6}x^3 - 238.25x$$



Some home practice exercises for students:

Problem-8: By definition, the singularity function can be expressed as follows:

$$\langle x-a\rangle^n = \begin{cases} (x-a)^n & \text{when } x > a \\ 0 & \text{when } x \le a \end{cases}$$

Develop an M-file that creates a plot of displacement (dashed line) versus distance along the beam, x. Note that $x = 0$ at the left end of the beam.

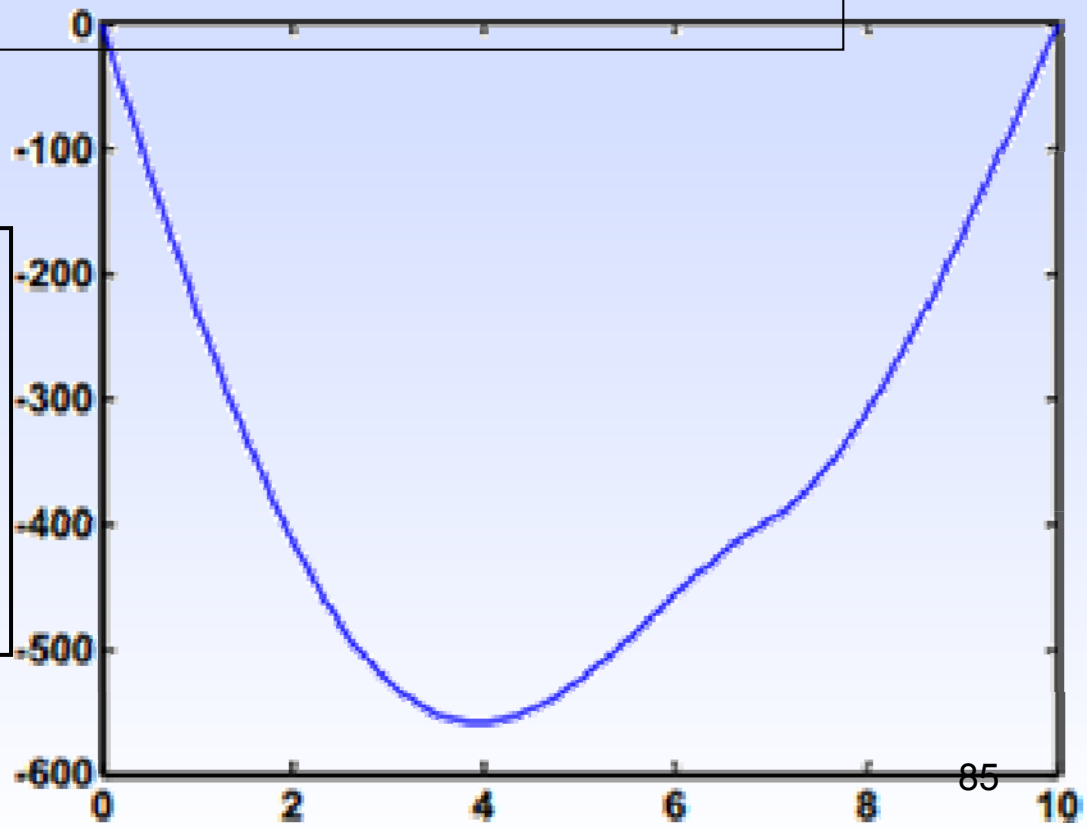
Solution is given in the next slide

P. 3.10

```
function beamProb(x)
xx = linspace(0,x);
n=length(xx);
for i=1:n
    uy(i) = -5/6.*(sing(xx(i),0,4)-sing(xx(i),5,4));
    uy(i) = uy(i) + 15/6.*sing(xx(i),8,3) + 75*sing(xx(i),7,2);
    uy(i) = uy(i) + 57/6.*xx(i)^3 - 238.25.*xx(i);
end
clf,plot(xx,uy,'--')
```

```
function s = sing(xxx,a,n)
if xxx > a
    s = (xxx - a).^n;
else
    s=0;
end
```

```
>> beamProb(10)
```



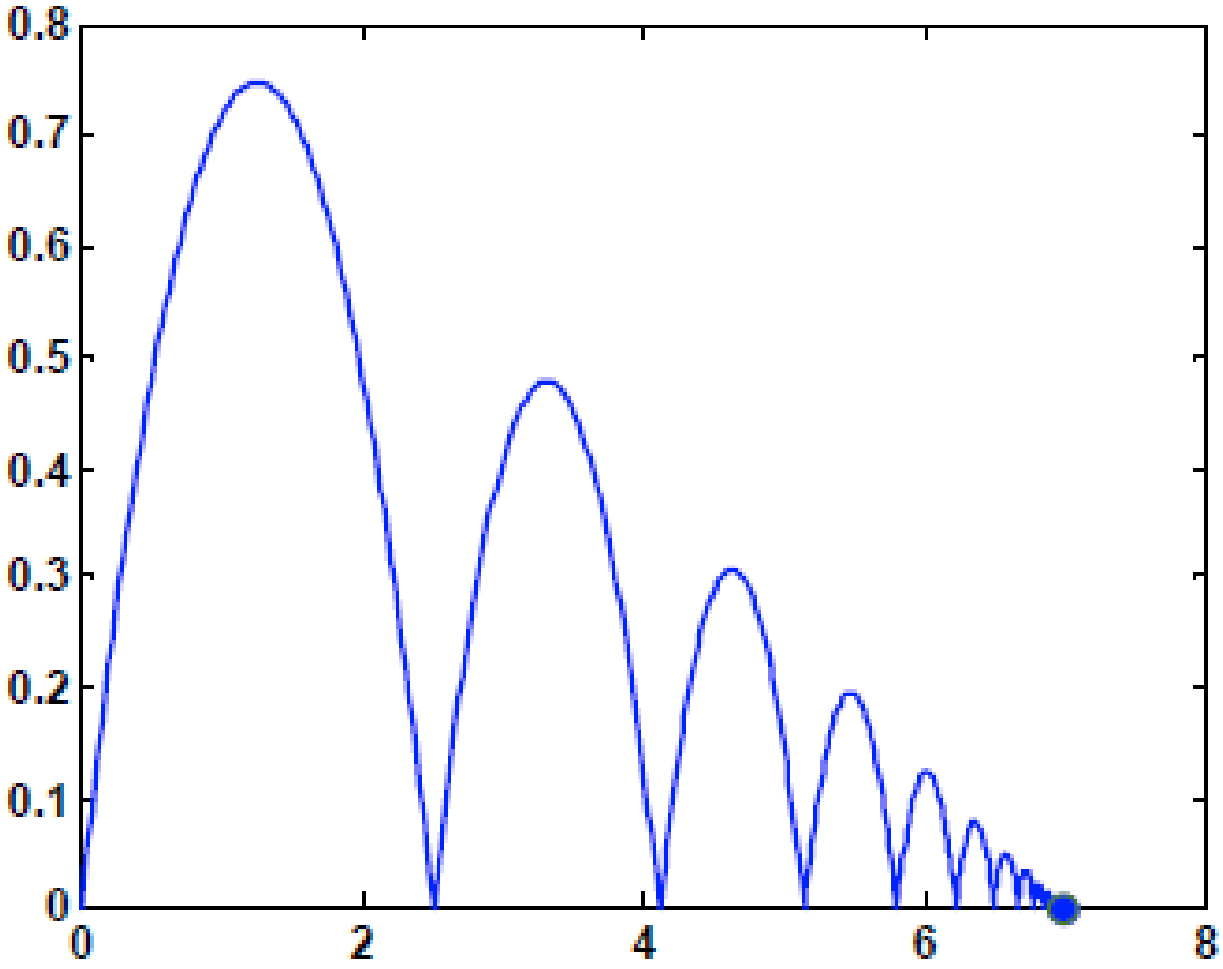
Some home practice exercises for students:

Problem-9:

3.21 Based on Example 3.6, develop a script to produce an animation of a bouncing ball where $v_0 = 5$ m/s and $\theta_0 = 50°$. To do this, you must be able to predict exactly when the ball hits the ground. At this point, the direction changes (the new angle will equal the negative of the angle at impact), and the velocity will decrease in magnitude to reflect energy loss due to the collision of the ball with the ground. The change in velocity can be quantified by the *coefficient of restitution* C_R which is equal to the ratio of the velocity after to the velocity before impact. For the present case, use a value of $C_R = 0.8$.

Solution is given in the next slide

```
clc,clf,clear
maxit=1000;
g=9.81; theta0=50*pi/180; v0=5; CR=0.83;
j=1;t(j)=0;x=0;y=0;
xx=x;yy=y;
plot(x,y,'o','MarkerFaceColor','b','MarkerSize',8)
xmax=8; axis([0 xmax 0 0.8])
M(1)=getframe;
dt=1/128;
j=1; xxx=0; iter=0;
while(1)
   tt=0;
   timpact=2*v0*sin(theta0)/g;
   ximpact=v0*cos(theta0)*timpact;
   while(1)
      j=j+1;
      h=dt;
      if tt+h>timpact,h=timpact-tt;end
      t(j)=t(j-1)+h;
      tt=tt+h;
      x=xxx+v0*cos(theta0)*tt;
      y=v0*sin(theta0)*tt-0.5*g*tt^2;
      xx=[xx x];yy=[yy y];
      plot(xx,yy,':',x,y,'o','MarkerFaceColor','b','MarkerSize',8)
      axis([0 xmax 0 0.8])
      M(j)=getframe;
      iter=iter+1;
      if tt>=timpact, break, end
   end
   v0=CR*v0;
   xxx=x;
   if x>=xmax|iter>=maxit,break,end
end
pause
clf
axis([0 xmax 0 0.8])
movie(M,1,36)
```


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8

Some home practice exercises for students:

Problem-10: 3.22 Develop a function to produce an animation of a particle moving in a circle in Cartesian coordinates based on radial coordinates. Assume a constant radius, r, and allow the angle, θ, to increase from zero to 2π in equal increments. The function's first lines should be

```
function phasor(r, nt, nm)
% function to show the orbit of a phasor
% r = radius
% nt = number of increments for theta
% nm = number of movies
```

Test your function with

```
phasor(1, 256, 10)
```

Solution is given in the next slide

3.22

```
function phasor(r, nt, nm)
% function to show the orbit of a phasor
% r = radius
% nt = number of increments for theta
% nm = number of movies
clc;clf
dtheta=2*pi/nt;
th=0;
fac=1.2;
xx=r;yy=0;
for i=1:nt+1
   x=r*cos(th);y=r*sin(th);
   xx=[xx x];yy=[yy y];
   plot([0 x],[0 y],xx,yy,':',...

x,y,'o','MarkerFaceColor','b','MarkerSize',8)
   axis([-fac*r fac*r -fac*r fac*r]);
   axis square
   M(i)=getframe;
   th=th+dtheta;
end
pause
clf
axis([-fac*r fac*r -fac*r fac*r]);
axis square
movie(M,1,36)
```

phasor(1,256,10)

Some home practice exercises for students:

Problem-11: **3.23** Develop a script to produce a movie for the butterfly plot from Prob. 2.22. Use a particle located at the x-y coordinates to visualize how the plot evolves in time.

2.22 The *butterfly curve* is given by the following parametric equations:

$$x = \sin(t)\left(e^{\cos t} - 2\cos 4t - \sin^5\frac{t}{12}\right)$$

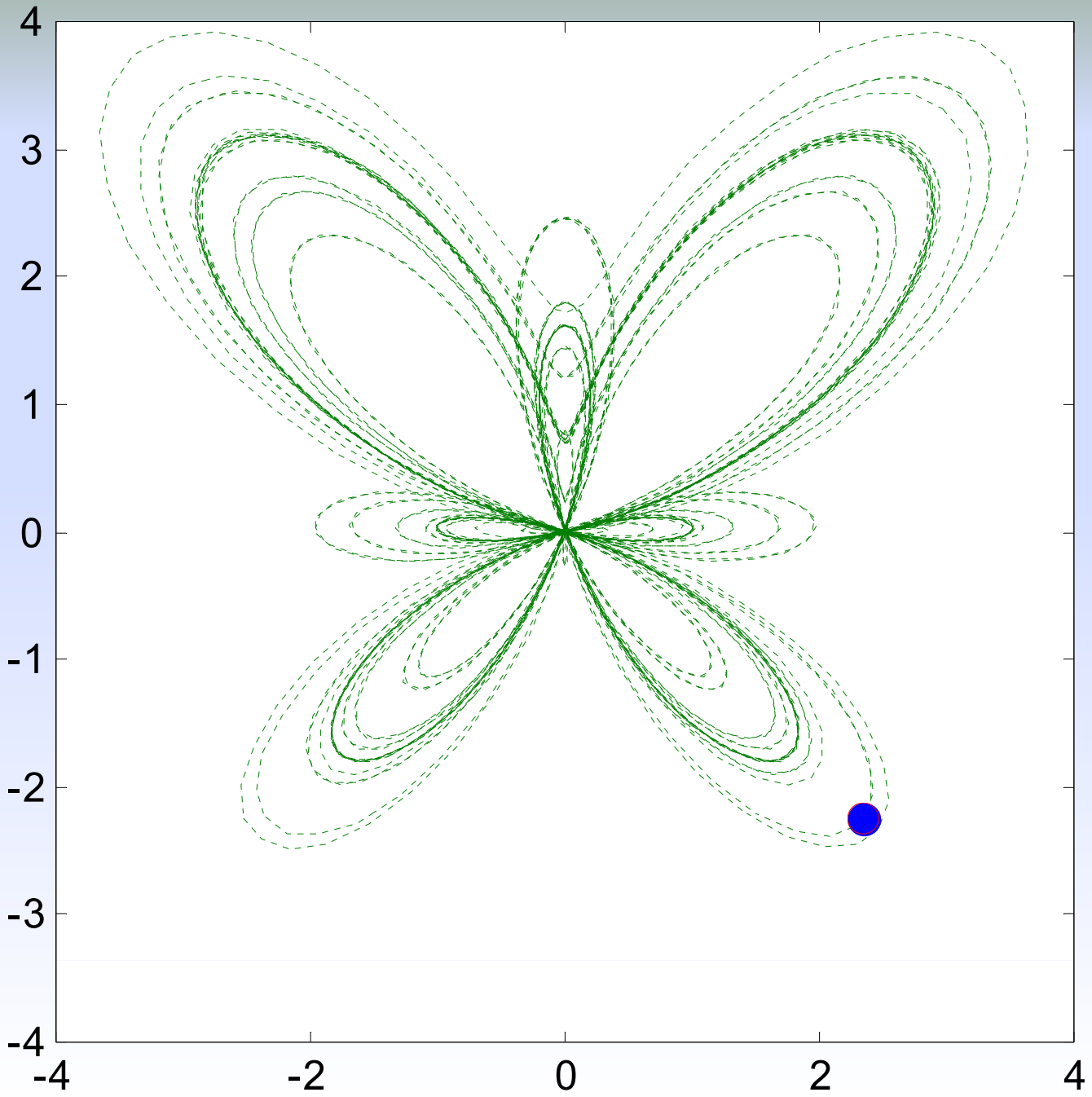
$$y = \cos(t)\left(e^{\cos t} - 2\cos 4t - \sin^5\frac{t}{12}\right)$$

Generate values of x and y for values of t from 0 to 100 with $\Delta t = 1/16$. Construct plots of **(a)** x and y versus t and **(b)** y versus x. Use `subplot` to stack these plots vertically and make the plot in **(b)** square. Include titles and axis labels on both plots and a legend for **(a)**. For **(a)**, employ a dotted line for y in order to distinguish it from x.

Solution is given in the next slide

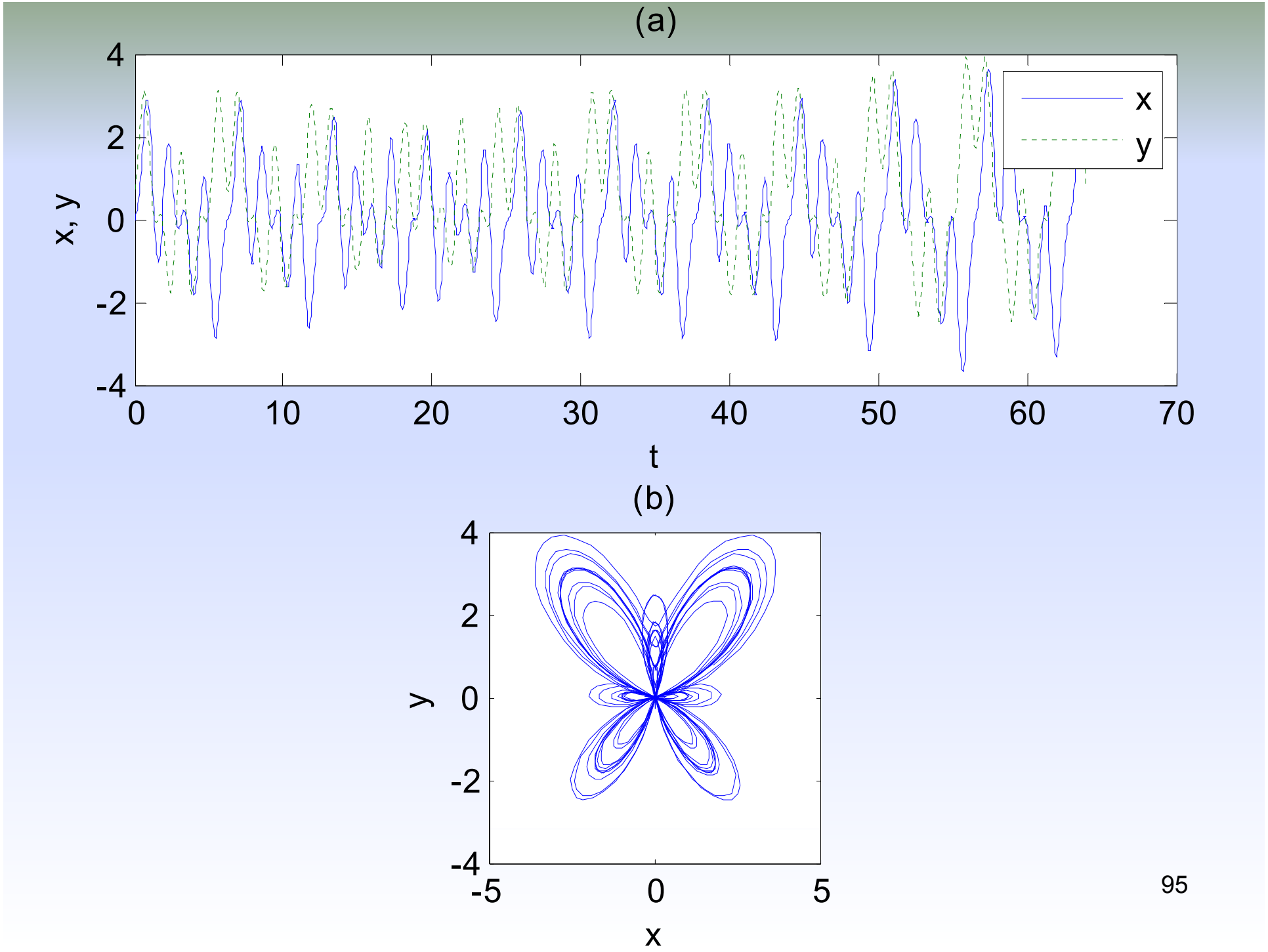
P3.23--Butterfly

```
clc;clf
t=[0:1/16:128];
x(1)=sin(t(1)).*(exp(cos(t(1)))-2*cos(4*t(1))-sin(t(1)/12).^5);
y(1)=cos(t(1)).*(exp(cos(t(1)))-2*cos(4*t(1))-sin(t(1)/12).^5);
xx=x;yy=y;
plot(x,y,xx,yy,':',x,y,'o','MarkerFaceColor','b','MarkerSize',8)
axis([-4 4 -4 4]); axis square
M(1)=getframe;
for i = 2:length(t)
    x=sin(t(i)).*(exp(cos(t(i)))-2*cos(4*t(i))-sin(t(i)/12).^5);
    y=cos(t(i)).*(exp(cos(t(i)))-2*cos(4*t(i))-sin(t(i)/12).^5);
    xx=[xx x];yy=[yy y];
    plot(x,y,xx,yy,':',x,y,'o','MarkerFaceColor','b','MarkerSize',8)
    axis([-4 4 -4 4]); axis square
    M(i)=getframe;
end
```


4
3
2
1
0
-1
-2
-3
-4
-4
-2
0
2
4

P2.22 Butterfly curves

```
clf
t=[0:1/16:64];
x=sin(t).*(exp(cos(t))-2*cos(4*t)-sin(t/12).^5);
y=cos(t).*(exp(cos(t))-2*cos(4*t)-sin(t/12).^5);
subplot(2,1,1)
plot(t,x,t,y,':');title('(a)');xlabel('t');ylabel('x, y');legend('x','y')
subplot(2,1,2)
plot(x,y);axis square;title('(b)');xlabel('x');ylabel('y')
```

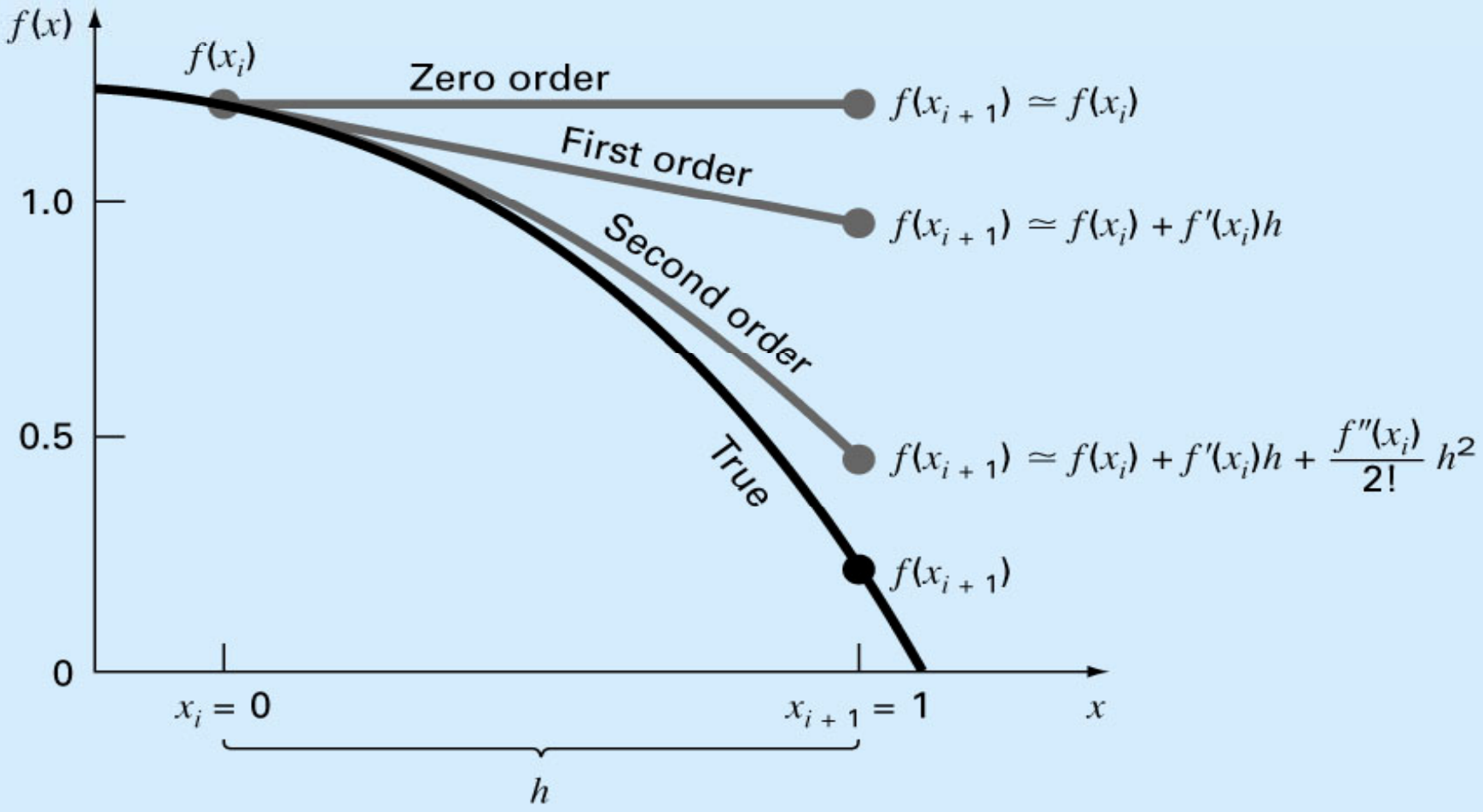

(a)
x, y
t
x
y
(b)
y
x

Truncation Errors and the Taylor Series

Chapter 4

- Non-elementary functions such as trigonometric, exponential, and others are expressed in an approximate fashion using Taylor series when their values, derivatives, and integrals are computed.
- Any smooth function can be approximated as a polynomial. Taylor series provides a means to predict the value of a function at one point in terms of the function value and its derivatives at another point.

FIGURE 4.1
The approximation of $f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$ at $x = 1$ by zero-order, first-order, and second-order Taylor series expansions.



Example:

To get the *cos*(x) for small x:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

If x=0.5

cos(0.5) =1-0.125+0.0026041-0.0000127+ …

=0.877582

From the supporting theory, for this series, the error is no greater than the first omitted term.

$$\therefore \quad \frac{x^8}{8!} \quad for \quad x = 0.5 \quad = 0.0000001$$

- Any smooth function can be approximated as a polynomial.

  $f(x_{i+1}) \approx f(x_i)$ *zero order* approximation, only true if x_{i+1} and x_i are very close to each other.

  $f(x_{i+1}) \approx f(x_i) + f'(x_i)\,(x_{i+1}-x_i)$ *first order* approximation, in form of a straight line

n^{th} *order* approximation

$$f(x_{i+1}) \cong f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''}{2!}(x_{i+1} - x_i)^2 + \ldots$$

$$+ \frac{f^{(n)}}{n!}(x_{i+1} - x_i)^n + R_n$$

$(x_{i+1}\text{-}x_i) = h$ *step size* (define first)

$$R_n = \frac{f^{(n+1)}(\varepsilon)}{(n+1)!} h^{(n+1)}$$

- Reminder term, R_n, accounts for all terms from (n+1) to infinity.

- ε is not known exactly, lies somewhere between $x_{i+1} > \varepsilon > x_i$.
- Need to determine $f^{n+1}(x)$, to do this you need $f'(x)$.
- If we knew f(x), there wouldn't be any need to perform the Taylor series expansion.
- However, $R=O(h^{n+1})$, $(n+1)^{th}$ order, the order of truncation error is h^{n+1}.
- O(h), halving the step size will halve the error.
- $O(h^2)$, halving the step size will quarter the error.

- Truncation error is decreased by addition of terms to the Taylor series.
- If *h* is sufficiently small, only a few terms may be required to obtain an approximation close enough to the actual value for practical purposes.

Example:

Calculate series, correct to the 3 digits.

$$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\ldots$$

- Truncation error is decreased by addition of terms to the Taylor series.
- If *h* is sufficiently small, only a few terms may be required to obtain an approximation close enough to the actual value for practical purposes.

Example:

Calculate series, correct to the 3 digits.

$$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\ldots$$

$\ln(1+x)$	$x-\frac{1}{2}x^2+\frac{1}{3}x^3-\frac{1}{4}x^4+\cdots$	$\sum_{n=0}^{\infty}\frac{(-1)^{n+1}}{n}x^n$

Error Propagation

- fl(x) refers to the floating point (or computer) representation of the real number x. Because a computer can hold a finite number of significant figures for a given number, there may be an error (round-off error) associated with the floating point representation. The error is determined by the precision of the computer (ε).

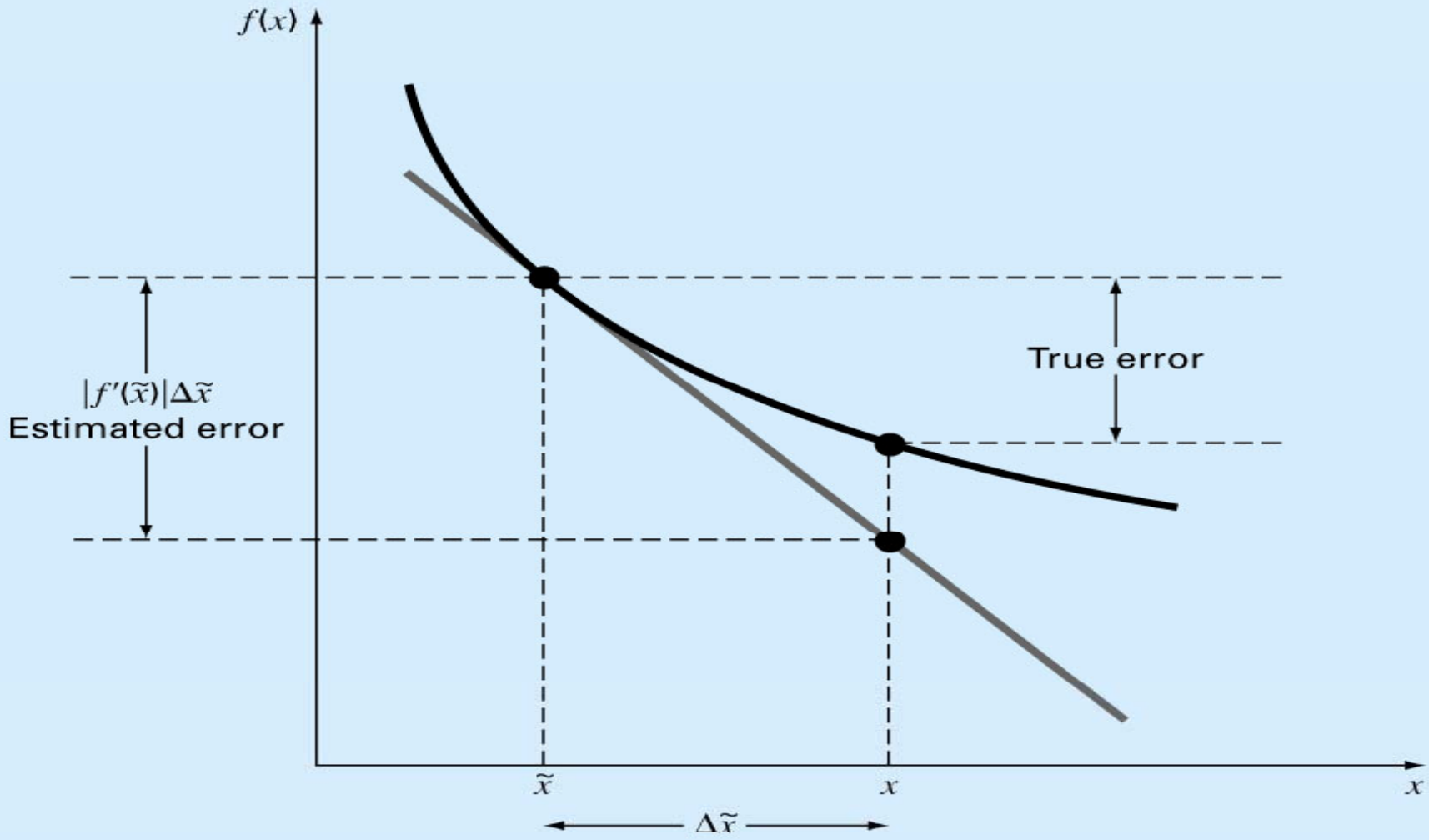
- Suppose that we have a function f(x) that is dependent on a single independent variable x. fl(x) is an approximation of x and we would like to estimate the effect of discrepancy between x and fl(x) on the value of the function:

$\Delta f(x_{fl}) = \left| f(x) - f(x_{fl}) \right|$ both f(x) and x_{fl} are unknown

Employ Taylor series to compute f(x) near f(x_{fl}), dropping the second and higher order terms

$$f(x) - f(x_{fl}) \cong f'(x_{fl})(x - x_{fl})$$

Figure 4.7



Also, let ε_t, the fractional relative error, be the error associated with $fl(x)$. Then

$$\frac{|fl(x) - x|}{|x|} = \varepsilon_t \quad where \quad \varepsilon_t \leq \varepsilon$$

Machine epsilon, upper boundary

Rearranging, we get

$$|fl(x) - x| = \varepsilon_t |x|$$

$$fl(x) = x(\varepsilon_t + 1)$$

- Case 1: Addition of x_1 and x_2 with associated errors ε_{t1} and ε_{t2} yields the following result:

  $fl(x_1)=x_1(1+\varepsilon_{t1})$

  $fl(x_2)=x_2(1+\varepsilon_{t2})$

  $fl(x_1)+fl(x_2)=\varepsilon_{t1}\, x_1+\varepsilon_{t2}\, x_2+x_1+x_2$

$$\frac{\varepsilon_t}{100\%}=\frac{fl(x_1)+fl(x_2)-(x_1+x_2)}{x_1+x_2}=\frac{\varepsilon_{t1}x_1+\varepsilon_{t2}x_2}{x_1+x_2}$$

•A large error could result from addition if x_1 and x_2 are almost equal magnitude but opposite sign, therefore one should avoid subtracting nearly equal numbers.

- Generalization:

  Suppose the numbers $fl(x_1)$, $fl(x_2)$, $fl(x_3)$, …, $fl(x_n)$ are approximations to x_1, x_2, x_3, … ,x_n and that in each case the maximum possible error is E.

  $$fl(x_i)\text{-}E \le x_i \le fl(x_i)+E \qquad E_{ti} \le E$$

  It follows by addition that

  $$\sum fl(x_i) - nE \le \sum x_i \le \sum fl(x_i) + nE$$

  So that

  $$-nE \le \sum x_i - \sum fl(x_i) \le nE$$

  Therefore, the maximum possible error in the sum of $fl(x_i)$ is nE .

- Case 2: Multiplication of x_1 and x_2 with associated errors e_{t1} and e_{t2} results in:

$$fl(x_1)\,fl(x_2) = x_1(1+\varepsilon_{t1})\,x_2(1+\varepsilon_{t2})$$

$$fl(x_1)\,fl(x_2) = x_1x_2(\varepsilon_{t1}\varepsilon_{t2} + \varepsilon_{t1} + \varepsilon_{t2} + 1)$$

$$\frac{\varepsilon_t}{100\%} = \frac{fl(x_1)\,fl(x_2) - x_1x_2}{x_1x_2} = \varepsilon_{t1}\varepsilon_{t2} + \varepsilon_{t1} + \varepsilon_{t2}$$

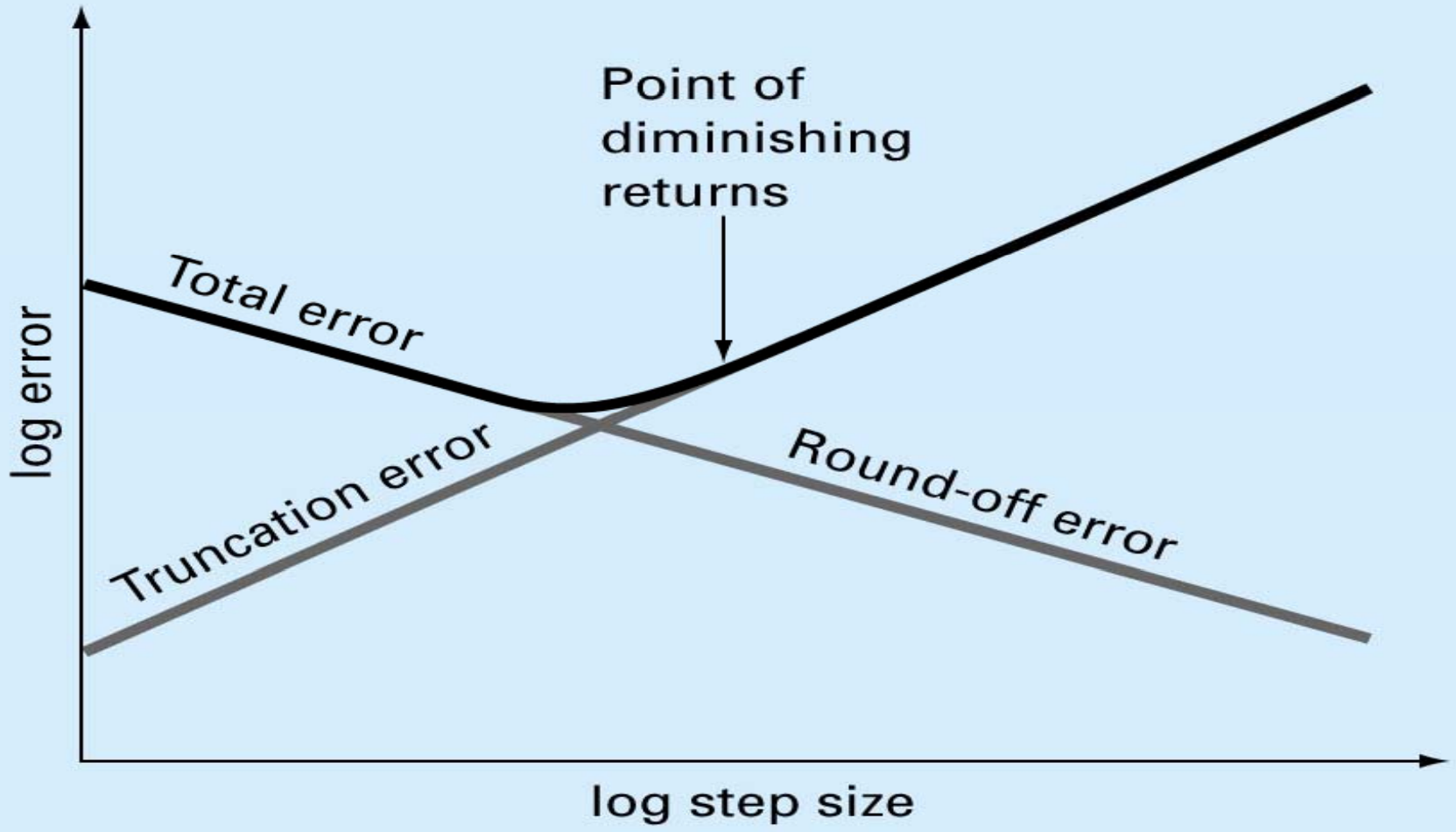
- Since ε_{t1}, ε_{t2} are both small, the term $\varepsilon_{t1}\varepsilon_{t2}$ should be small relative to $\varepsilon_{t1}+\varepsilon_{t2}$. Thus the magnitude of the error associated with one multiplication or division step should be $\varepsilon_{t1}+\varepsilon_{t2}$.

  $\varepsilon_{t1} \leq \varepsilon$ (upper bound)

- Although error of one calculation may not be significant, if 100 calculations were done, the error is then approximately 100ε. The magnitude of error associated with a calculation is directly proportional to the number of multiplication steps.
- Refer to Table 4.3

- Overflow: Any number larger than the largest number that can be expressed on a computer will result in an overflow.
- Underflow (Hole) : Any positive number smaller than the smallest number that can be represented on a computer will result an underflow.
- Stable Algorithm: In extended calculations, it is likely that many round-offs will be made. Each of these plays the role of an input error for the remainder of the computation, impacting the eventual output. Algorithms for which the cumulative effect of all such errors are limited, so that a useful result is generated, are called “stable” algorithms. When accumulation is devastating and the solution is overwhelmed by the error, such algorithms are called unstable.

Figure 4.8


Point of
diminishing
returns
Total error
log error
Truncation error
Round-off error
log step size