Introduction to Computing with MATLAB

Arun Prakash

School of Civil Engineering
Purdue University.

Contents

Chapter 1

Introduction to Computing

Using computers to solve (engineering) problems of our interest is called Computing. In this process, we develop *computational tools* that help us do our jobs better and faster. Computing is different from Computer Science. Computer Scientists try to design the Computer itself and develop programming languages that we, as programmers, can use for our own engineering applications.

1.1 Computing

Why do we need Computing?

- Volume of data and societal needs have grown beyond human capabilities
- Human error, consistency of results, speed and accuracy
- Examples: Banking, Automotive, Manufacturing, Communication etc.
- Questions: Reliability, Fault tolerance, robustness, backup
- Caveat: Utilization vs. Dependence; we are responsible for the technology we create and use.

Types of Computing

- On-site Data Analysis and response systems in real-life applications:
  Structural Health Monitoring and Control
  Water quality management
  Earthquake Engineering

- Direct simulation of physical phenomena (Scientific Computing)
  Analysis & design of systems such as buildings, bridges, machines etc.
  Verify & Validate current and future theories of physics - Simulate stuff we cannot measure or observe - subatomic particles, core of stars, even origin of the universe!!

Components of a computer

- Hardware
  CPU - Binary (0,1) instructions go in ; Binary output obtained
  Memory - ROM, RAM, Hard Disk, External Storage
  Input Devices - Keyboard, Mouse, Touch Screen etc.
  Output Devices - Monitor, Printer etc.

- Software

  - Operating system
    Windows, MacOS, Linux, Unix, Sun Solaris,
  - Applications Programs
    Internet Explorer, Media Players, Photoshop, Adobe Acrobat
    Programming languages: C/C++, Fortran, Pascal etc.
    MATLAB
    Your programs

1.2 Computer Programming

What is programming

Defining the set of operations for a computer to perform (telling the computer what to do). Computers understand only certain binary instructions. So computer scientists developed more user friendly *languages* that translated (compiled / interpreted) into binary code. We need to learn these languages in order to communicate with the computers by writing *programs*.

- Structure of a program:

  - Get input
  - Compute - operate upon the input data to generate meaningful information
  - Output the results

- Some Essentials of Programming

  - Data Structure for Memory management - Variables, Array, Pointers
  - Conditional Branching Statements
  - Loops
  - File Handling
  - Input / Output
  - Graphics

MATLAB - Matrix Laboratory

Advantages:

- Relatively easy to use and good for beginners, GUI
- Predefined functions for a lot of Mathematical operations:
  Matrix Algebra, Solving system of equations, Eigenvalue computations
- Symbolic Mathematics: Algebra, Differentiation, Integration
- Additional Toolboxes
- Plotting / Imaging / Visualization of Results - device independent
- Combine Languages, C/C++, Fortran
- Different platforms run the same MATLAB program / code
- Demos : Membrane, 3D peaks, Bar with notch

Disadvantages:

- Interpreter based : Slower, but can be compiled
- Kernel overhead - not suitable for very large problems
- Limited advanced programming features:
  Pointers, Pass by Reference, Object-oriented

The MATLAB Environment

- Desktop
- Command window
  $\gg$ is called the 'command prompt'
  Arithmetic: `+, -, *, /,` $\wedge$
  Line continuation (Ellipsis) ...
- Command History window
- Workspace browser: Variables `whos, clear, clc, clf`
- Path Browser - Variable, m-file in current directory, first occurrence
  `which`
- Editor window : m-files as scripts
- Figure windows
  `plot`

- Help
  `help, lookfor`
  Getting started section
- Start Button
- Other commands
  CTRL-c : Cancel or Interrupt Operation (when MATLAB 'hangs')
  ! : execute command on MS-DOS or Unix *shell* prompt
  `diary`

Built-in functions in MATLAB

Elementary Math Functions:

- abs( )
- sqrt( )
- factorial( )
- exp( )
- log( ) ; log10( )

Trigonometric functions

- sin( ) ; asin( )
- cos( ) ; acos( )
- tan( ) ; atan( )
- cot( ) ; acot( )

Hyberbolic functions

- sinh( ) ; cosh( )
- tanh( ) ; coth( )

1.3 Basic Matrix Algebra

Refer to Matrices Handout.

Chapter 2

MATLAB Basics: Datatypes, Arrays, Input/Output, Plotting

Before we can write programs, it is important to understand how MATLAB uses and operates on different *types* of data.

2.1 Datatypes in MATLAB

The two most common data types in MATLAB are **Numeric** and **character** data (Refer to MATLAB help for details on other types of data).

1. Numeric Data is stored in `double` precision format by default. Double precision numbers use 64 bits (binary digits - 0, 1) and can store a number with 15 to 16 significant digits of precision (mantissa) and 10^{-308} to 10^{308} as exponent. Double precision data types can be **real**, **imaginary** or **complex**.

2. Character data types are stored in 16-bit value representing a single character. **Strings** are a collection of characters where each character uses 16 bits.
   Example `char(65)` is 'A' and `char(97)` is 'a'.

2.1.1 Variables

A **Variable** is user given name that refers to a certain location in the computers memory where MATLAB stores data. The user can access that data by specifying the variable name associated with it.
Rules:

1. Variable names are **case sensitive**. Example: `var, Var, VAR` are all different.

2. Must begin with an **alphabet** followed by alphabets, numbers and the underscore _ character.

3. MATLAB can distinguish variable names upto 63 characters in length.

Examples

```
x = 10
x = x + 1
X = 20 + 20i
character1 = 'a'
character2 = '1'
CharVar1 = char(97)
StrVar1 = 'This is CEE 15'
string_variable = 'That''s cool!'
```

Predefined Variables in MATLAB (not protected: can be overwritten)

```
pi
i, j
Inf, Nan, ans
realmax, realmin, eps
clock, date
```

Note: Choose the names of your variables so that no inbuilt predefined variables or functions are over-written.

2.1.2 Arrays

MATLAB treats **all** data as **arrays**. An **array** is a 'collection of data' (any data - numbers, characters etc.) that is stored in continuous locations in the computers memory. All variables refer to arrays in the computers memory. Even scalars are actually treated as 1×1 array.

Arrays are primarily of two types: **Vectors** (dimension 1) and **Matrices** (2 or more dimensions). The **size** of an array is the number of rows and columns in an array. (For higher dimensional arrays it includes the extent of all dimensions).

Example:

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad 3 \times 2 \text{ matrix}$$

$$b = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \qquad 1 \times 4 \text{ array, } \textbf{row vector}$$

$$c = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad 3 \times 1 \text{ array, } \textbf{column vector}$$

COMMANDS: `length(), size()`

`size(a)` gives the size of a specific matrix `a`.

`length(a)` returns the length of a vector or the longest dimension of a 2-D array.

Individual **elements** in an array are accessed using the row and column number of the element in **parentheses**. For example, in the above arrays `a(2,1)` is 3, `b(2)` is 2, and `c(3)` is 3.

2.1.3 Initialization of Variables and Arrays

Variables need not be declared prior to using them (unlike C, C++, Fortran etc.). Variables can be created and stored using:

1. **Assignment**

   ```
   var = expression
   area = pi*(2.3)^2
   myarray1 = [1 2 3 ; 4 5 6 ]
   myarray1(3,2) = 1              (expanding an existing array)
   ```

   Note If a particular subscript in not in range of an array, MATLAB automatically increases the dimensions of the array to fit the new element.

2. **Shortcut Expressions**

   ```
   var = first :  inc :  last              (default inc is 1)
   myarray2 = [1:5]              creates a row vector
   myarray3 = [ 1:5:26 ; 25:5:50 ]
   ```

   Note: Number of entries in each row must be equal.

3. **Combining arrays**

   ```
   col1 = [1:3]'
   col2 = [6:-1:4]'
   myarray4 = [ col1 col2 ]
   myarray4 = [ myarray4 col2 col1 ]
   name = ['Mike' ' ' 'Smith']
   ```

4. **Built-in Functions**

   ```
   magic( ), zeros( ), ones( ), eye( )
   ```

   magic: The `magic(n)` function generates an $n \times n$ matrix constructed from the integers from 1 through n^2. The integers are ordered in such a way that all the row sums and all the column sums are equal to the same number.
   zeros: The `zeros` function generates a matrix containing all zeros.
   ones: The `ones` function generates a matrix containing all ones.
   eye: The `eye` function generates an identity matrix.

Summary of symbols related to array operations

Character	Description
:	Used in short-cut expressions
=	Assignment operator
( )	Subscripts of arrays
[ ]	Brackets; forms arrays
,	Separates array elements
;	Semicolon; suppresses echo of input, ends row in array
'	Single quote; matrix transpose, creates string

2.1.4 Multi-dimensional Arrays

Three dimensional arrays can be visualized as cuboids and can be addressed using 3 subscripts. For example

```
array3d(:,:,1) = [1 2 3 ; 4 5 6]
array3d(:,:,2) = [7 8 9 ; 10 11 12]
```

is a $2 \times 3 \times 2$ array.

However higher dimension arrays are harder to visualize and should be thought of in terms of subscripts. For example

```
array4d(2,2,2,2) = 1
```

is a $2 \times 2 \times 2 \times 2$ array.

2.1.5 Subarrays

It is possible to select and use subsets of MATLAB arrays as though they were separate arrays. To select a portion of an array, just include a list of all the elements to be selected in the parentheses after the array name. For example,

```
arr1 = [1.1 -2.2 3.3 -4.4 5.5];
arr1(3)          ⟶          3.3
arr1([1 4])      ⟶          [1.1 -4.4]
arr1([1:2:5])    ⟶          [1.1 3.3 5.5]
```

For a two-dimensional array, a colon can be used in a subscript to select all of the values of that subscript. For example,

```
arr2 = [1 2 3; -2 -3 -4; 3 4 5];
arr2(1,:)          ⟶          [1 2 3]
arr2(:,1:2:3)      ⟶          [1 3; -2 -4; 3 5]
```

The **end** function
`end` function returns the highest value taken on by that subscript, For example

```
arr2(2:end,:)      ⟶          [-2 -3 -4; 3 4 5]
```

Assigning using subarrays
Subarrays can also be used to change the values of that portion of the main array. For example,

```
arr2(:,1:2:3) = [111 222 ; 333 444 ; 555 666 ]
arr2(:,1:2:3) = 10
```

Empty array [ ] ; Deleting elements of an array
Elements of an array can be deleted by assigning them to the empty array `[]`.

```
arr3 = magic(7)
arr3([1 3],:)  = []
```

2.2 Matrices Operations vs. Arrays Operations

2.2.1 Matrix operations

MATLAB has all the operators of conventional matrix algebra already built in.

Addition and Subtraction of Matrices is carried out on two or more matrices of the *same size* by adding or subtracting the corresponding elements of the matrices.

Transpose of a Matrix The transpose of a matrix is a new matrix in which the rows of the original matrix are the columns of the new matrix. If a matrix contains a complex value then we can have both the complex conjugate transpose (`ctranspose` and `'`) and complex nonconjugate transpose (`transpose` and `.'`).

Dot Product MATLAB command: `c = dot(a,b)`
The dot product is the scalar computed from two vectors of the same size.

$$c = \sum_{i=1}^{n} a_i b_i$$

Matrix Multiplication MATLAB command: `c = a*b`
The matrix multiplication is defined by

$$c = a * b \qquad c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

For example if matrices `a` and `b` of dimensions $m \times n$ and $n \times p$ respectively are such that number of *columns* of `a` are equal to number of *rows* in `b` (in this case: n) then the resulting matrix `c` will have dimensions $m \times p$ according the above formula.

Matrix Powers The command for the power of a matrix `a` is `a^2` (where, power is equal to 2). `a^2` is equivalent to `a*a`. Similarly, `a^4` is equivalent to `a*a*a*a`. To raise a matrix to a power, the matrix must be a square matrix.

Matrix Inverse MATLAB Command: `b = inv(a)`
By definition, if `b` is an inverse of a square matrix `a`, then `a*b` or `b*a` are both equal to an identity matrix with only the diagonal elements being 1 and other elements being 0.

Determinants MATLAB Command: `det(a)`

Solving system of equations
The solution of a system of equations $A\,x = b$ is given by $x = A^{-1}\,b$. The direct way of calculating this solution using `x = inv(A)*b` is *expensive*. Alternatively, MATLAB can solve this system using *Gaussian elimination* which is implemented as the **backslash** \ .
MATLAB Command: `x = A \ b`.

2.2.2 Array operations

Sometimes we have to perform arithmetic operations between the elements of two arrays of the **same size** in an **element-by-element** manner. These operators are denoted by **preceding** the normal arithmetic operators by a **dot .** such as (`.+, .-, .*, ./, .^`) . For example if `a` and `b` are matrices of same size:

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

`a .* b` denotes element-by-element multiplication of `a` and `b`. A normal matrix multiplication between the above matrices is not defined.

Note + & .+ - & .- operations produce the exact same result.

Summary of Array and Matrix operators

Character	Description
+ or -	Array **and** Matrix addition or subtraction of arrays
.*	Element-by-element multiplication of arrays
./	Element-by-element right division : a/b = `a(i,j)/b(i,j)`
.\	Element-by-element left division : a\b = `b(i,j)/a(i,j)`
.^	Element-by-element exponentiation
*	Matrix multiplication
/	Matrix right divide : `a/b = a*(b)`$^{-1}$
\	Matrix left divide (equation solve) : `a\b = (a)`$^{-1}$ `* b`
^	Matrix exponentiation

Precedence (higher to lower):

1. Parentheses ( )
2. transpose .', power .^, complex conjugate transpose ', matrix power ^
3. unary operator: Unary plus +, unary minus -, logical negation ~
4. multiplication .*, right division ./, left division .\, matrix multiplication *, matrix right division /, matrix left division \
5. addition +, subtraction -
6. colon operator :

For the operators with the same precedence, the executions proceed from left to right.

Refer to MATLAB help for complete precedence rules
MATLAB → Programming → Basic Programming Components → Operators

2.3 Input and Output (I/O) of Data

2.3.1 Input the data from keyboard

We can ask the user to provide input data using the `input()` command.

```
var = input('Enter the value to be stored:  ')
```

This allows the user to enter any valid MATLAB expression, that evaluates to a *numeric* or *character* value.

```
stringvar = input('Enter the string to be stored:  ','s')
```

When used with the option `'s'`, anything that the user enters is stored as character data.

2.3.2 Output of Data to the Screen

The `format` statement

In MATLAB the decimal fractions are printed using a default format (short format) that shows 4 decimal decimal digits (eben though MATLAB internally stores double precision variables with 14-15 digits of accuracy). If we want values to be displayed in a decimal format with 14 decimal digits, we use the command `format long`. The format can be returned to a decimal format with 4 decimal digits using the command `format short`. `format short e` command will print the values in scientific notation with 5 significant digits and `format long e` prints the same but with 15 significant digits. `format +` command is used to print the sign only. When a matrix is printed with the `format +` command, the only character printed is plus and minus signs. If a value is positive a plus sign will be printed; if a value is zero, a space will be printed; if a value is negative, a minus sign will be printed.

Display formats

Command	Description
`format short`	(Default) Fixed-point format with 4 decimal digits
`format short e`	Scientific notation with 4 decimal digits
`format short g`	Best of 5-digit fixed or floating point
`format long`	Fixed-point format with 14 decimal digits
`format long e`	Scientific notation with 15 decimal digits
`format long g`	Best of 15-digit fixed or floating point
`format bank`	Two decimal digits
`format compact`	Eliminates empty lines
`format loose`	Adds empty lines
`format +`	Only signs are printed

The `disp` and `num2str` functions

The `disp` function accepts **one** array argument whether numeric or string, and displays the value of the array in the Command Window. If the array is of type `char`, then the character string contained in the array is printed out.

This is useful for displaying the final result of a program. `num2str` function can be used to convert numeric values to character strings and then use `disp()` for displaying them. For example,

```
n = 20;
disp(['Total number of students in the class =' num2str(n)])
disp(n)
```

The `fprintf` Function

The general form of the `fprintf` function is:

```
fprintf (format, data)
```

where `format` is a string that controls the way the data is to be printed, and `data` is one or more scalars or arrays to be printed. The `format` is a character string containing text to be printed plus special characters describing the format of the data.

Common Special Characters in `fprintf` Format Strings

Format String	Results
%d	Display value as an integer
%e	Display value in exponential format
%f	Display value in floating point format
%g	Display value in either floating point or exponential format, whichever is shorter
%c	Display a single character
%s	Display a string of characters
\n	Skip to a new line

Example

```
temp = 78.234567989;
fprintf('The temperature is %f degrees.  \n',temp)
```

will print: `The temperature is 78.2346 degrees.`

It is also possible to specify the width of the field in which a number will be displayed and the number of decimal places to display. This is done by specifying the width and precision after the `%` sign and before the `f`. For example,

```
fprintf('The temperature is %4.1f degrees.  \n',temp)
```

will print: `The temperature is 78.2 degrees.`

The output contains the value of `temp` printed with 4 positions, one of which will be a decimal position as shown above.

2.3.3 I/O through Data Files

Matrices can be defined from information that has been stored in a data file. MATLAB can interface to two different types of data files.

- **MAT files**: Data stored in a memory-efficient binary format. They are preferable for data that are going to be generated and used by MATLAB programs only.
- **ASCII files**: Data stored in ASCII characters. They are necessary if the data are to be shared (imported or exported) with programs other than MATLAB.

MAT Files

`save filename var1 var2 var3`
The `save` command saves the values of variables `var1, var2,` etc. in a file named `filename`. By default, the file name will be give the extent `mat`, and such data files are called MAT-files.

`save filename x y z -append`
adds the variables `x, y, z` to an existing MAT file `filename.mat`.

The **load** command is the opposite of the **save** command. It loads data from a disk file into the current MATLAB workspace. For example,
`load filename`

ASCII Files

- The ASCII files must contain only numeric information. We can use % in the ASCII file for comment lines.
- Each row of the ASCII file must contain the same number of data values to be read by another program in MATLAB.

```
save temp.dat c d -ascii
```

- By loading the ascii file, data value will be automatically stored in a matrix `temp` (with the same name as the data file) which will have the same size as the data.
- Though values of the variables c and d were stored in the temp.dat file, they can be read as a variable matrix (`temp`) for our case.

```
load temp.dat
```

2.4 Introduction to Plotting

Plotting is useful when we have to display the output / results of our program in a graphical format.

2.4.1 The `plot` command

The `plot()` command generates an x-y plot using 2 arrays. For example to plot the function $y = x^2 - 10x + 15$ for values of x between 0 and 8.

```
x = 0:1:8;
y = x.^2 - 8.*x + 15;
plot(x,y);
```

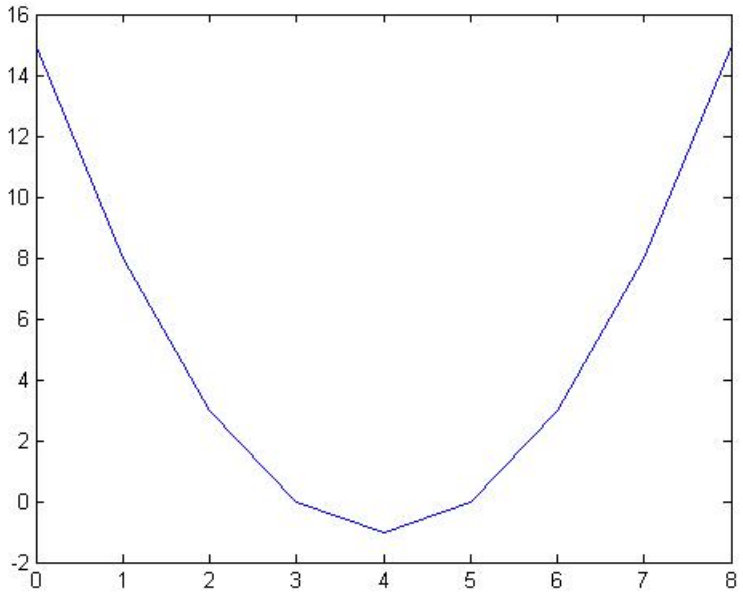

Figure 2.1: Plot of $y = x^2 - 8x + 15$ from 0 to 8.

The first statement creates a vector of x values between 0 and 10 using the colon operator. The second statement calculates the y values from the equation. Finally, the third statement creates the plot using `plot` function. When the `plot` function is executed, Matlab opens a Figure Window and displays the plot in the window, see Figure 2.1.

Note: Both vectors of `x` and `y` must have the same length.

2.4.2 Title, Label, Grid and Text

Titles and axis labels can be added to a plot with the `title`, `xlabel`, and `ylabel` functions. Each function is called with a string containing the title or label to be applied to

the plot. Grid lines can be added or removed from the plot with the `grid` command: `grid on` turns on the grid lines, and `grid off` turns off grid lines.

For example, titles, labels and grid lines are applied to the previous figure, as shown in Figure 2.2.

```
x = 0:1:8;
y = x.^2 - 8.*x + 15;
plot(x,y);
title('Plot of y = x^2 - 8*x + 15');
xlabel('x'); ylabel('y');
grid on;
```

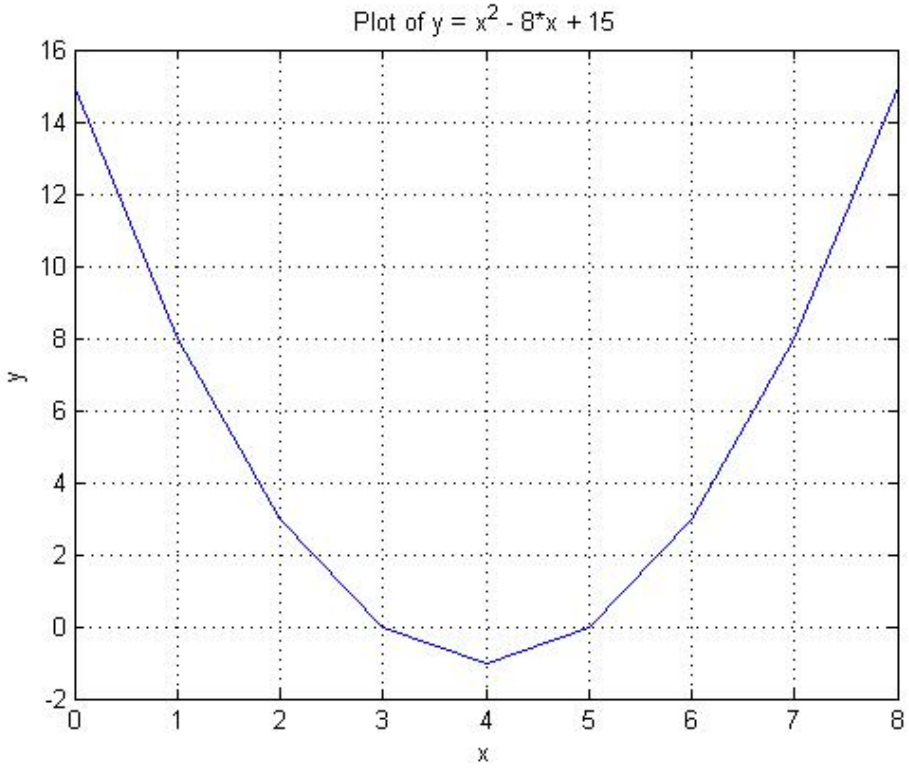

Figure 2.2: Plot of $y = x^2 - 8x + 15$ from 0 to 8 with a title, axis labels, and grid lines.

The function, `text(x,y,'string')` writes the `string` on the graphics screen at the points specified by the coordinates `(x,y)` using the axes from the current plot. If `x` and `y` are vectors then the text is written in each point.

```
text{2.5, 3, 'y(x) = x^2 - 8x + 15'}
text{2.1, 3, '\leftarrow'}
```

2.4.3 Multiple curves on one plot

It is possible to plot multiple curves on the same graph. We can plot multiple curves on the same graph by using multiple arguments in the plot command, for example, `plot(x1,y1,x2,y2)`. Here, `x1`, `y1`, `x2` and `y2` are vectors. When the command is executed, the curve corresponding to `x1` and `y1` will be plotted, and then the curve corresponding to `x2` and `y2` will be plotted on the same graph. Another way to plot multiple curves on the same graph is with the `hold` command. After a `hold on` command is issued, all additional plots will be laid on top of the previously existing plots. A `hold off` command switches plotting behavior back to the default situation, in which a new plot replaces the previous one.

For example, suppose that we want to plot the function $f(x) = sin2x$ and its derivative, $2cos2x$, on the same plot. We can use either of the following two ways (the result is shown in Figure 2.3):

```
x = 0:pi/100:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x,y1,x,y2);
```

or

```
x = 0:pi/100:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x,y1);
hold on;
plot(x,y2);
```

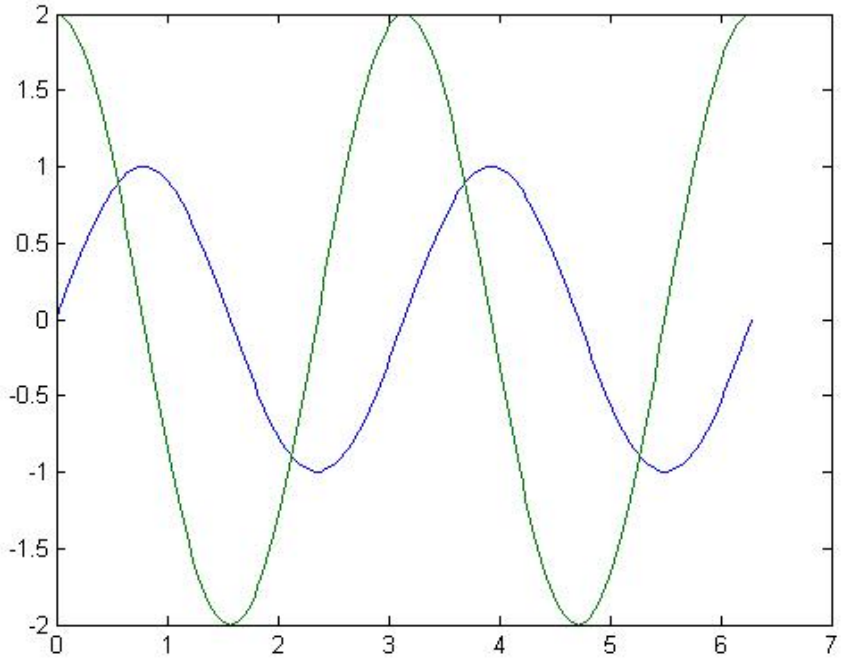

Figure 2.3: Plot of $f(x) = sin2x$ and $f(x) = 2cos2x$ on the same graph.

2.4.4 Line Color, Line Style, Marker Style, and Legends

Matlab allows to select the color of a line to be plotted, the style of the line to be plotted, and the type of marker to be used for data points on the line. These traits may be selected using an attribute character string after the x and y vectors in the plot function.

The attribute character string can have up to three characters, with the first character specifying the color of the line, the second character specifying the style of the marker, and the last character specifying the style of the line. The characters for various colors, markers, and line styles are shown in the following table:

Table of Plot Colors, Marker Styles, and Line Styles

Color		Marker Style		Line Style	
`y`	yellow	`.`	point	`-`	solid
`m`	magenta	`o`	circle	`:`	dotted
`c`	cyan	`x`	x-mark	`-.`	dash-dot
`r`	red	`+`	plus	`--`	dashed
`g`	green	`*`	star	`<none>`	no line
`b`	blue	`s`	square		
`w`	white	`d`	diamond		
`k`	black	`v`	triangle (down)		
		`^`	triangle (up)		
		`<`	triangle (left)		
		`>`	triangle (right)		
		`p`	pentagram		
		`h`	hexagram		
		`<none>`	no marker		

The attribute characters may be mixed in any combination, and more than one attribute string may be specified if more than one pair of (x, y) vectors are included in a single `plot` function.

Enhanced Control Plotted Lines

It is also possible to set additional properties associated with lines and markers in the figure.

- `LineWidth` specifies the width of each line in points.
- `MarkerEdgeColor` specifies the color of the marker or the edge color for filled markers.
- `MarkerFaceColor` specifies the color of the face of filled markers.
- `MarkerSize` specifies the size of the marker in points.

These properties are specified in the `plot` command after the data to be plotted in the following fashion:

```
plot(x, y, 'PropertyName', value, ...)
```

Adding Legends
Legends may be created with the `legend` function. The basic form is

```
legend('string1', 'string2', ..., 'Location', pos)
```
or
```
legend('string1', 'string2', ...)
```
where `string1`, `string2`, etc. are the labels associated with the lines plotted, and `pos` may be a string specifying where to place the legend. Command `legend off` will remove an existing legend. The possible values of `position` are given in the following table.

Values of `position` in the `legend` Command

Value	Legend Location
`'NorthWest'`	Top left corner
`'North'`	Top center
`'NorthEast'`	Top right corner (default)
`'West'`	Middle left edge
`'East'`	Middle right edge
`'SouthWest'`	Bottom left corner
`'South'`	Bottom center
`'SouthEast'`	Bottom right corner

2.4.5 Controlling x- and y-axis Plotting Limits

By default, a plot is displayed with x- and y-axis ranges wide enough to show every point in an input data set. However, we can use the `axis` command/function to control axis scaling and appearance. Some of the forms of the `axis` command/function are:

1. `v = axis;` returns a 4-element row vector containing the current limits, `xmin, xmax, ymin`, and `ymax`, of the plot.
2. `axis([xmin xmax ymin ymax])` sets the x and y limits of the plot to the specified values.
3. `axis equal` sets the axis increments to be equal on both axes.
4. `axis square` makes the current axis box square.
5. `axis normal` cancels the effect of axis equal and axis square.
6. `axis off` turns off all axis labelling, tick marks, and background.
7. `axis on` turns on all axis labelling, tick marks, and background (default case).

An example of a complete plot is shown in Figure 2.4, and the statements to produce that plot are shown below.

```
x = 0:pi/25:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x, y1, 'go-', 'MarkerSize', 6.0, 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'g');
hold on;
plot(x, y2, 'rd-', 'MarkerSize', 6.0, 'MarkerEdgeColor', 'r', 'MarkerFaceColor', 'g');
title('Plot of f(x) = sin(2x) and its derivative');
xlabel('x');
ylabel('y');
legend('f(x)', 'd/dx f(x)', 'Location', 'NorthWest');
```

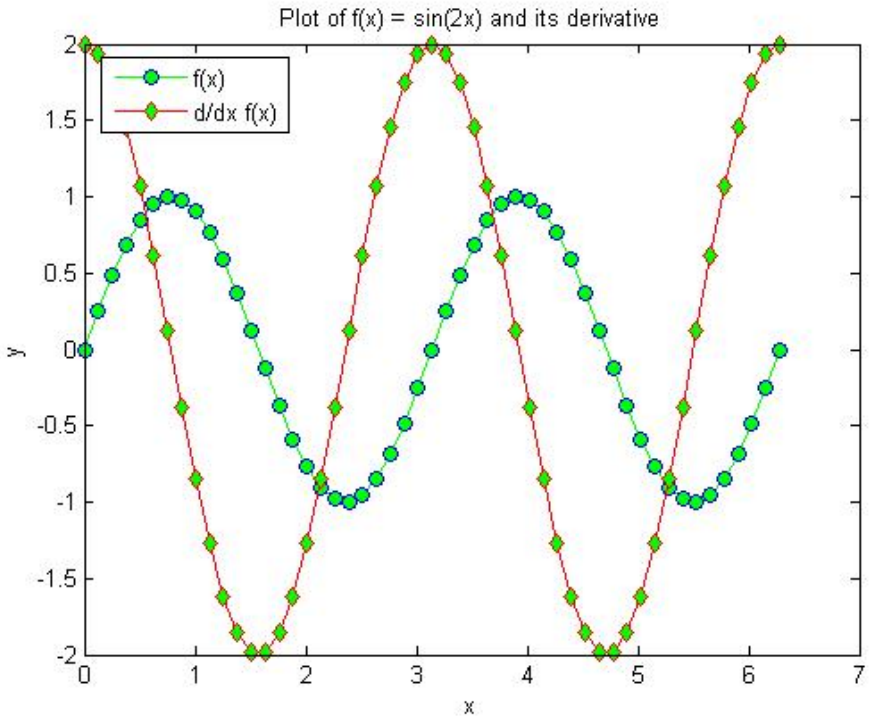

Figure 2.4: A complete plot with title, axis labels, legend, grid, and multiple line styles.

2.4.6 Controlling Plot features using the GUI

You can perform similar operations to control the settings of a plot using GUI *Plotting Tools*. However, this process has to be done manually and is not recommended when dealing with a large of plots. Refer to MATLAB help for details on plotting tools.

Chapter 3

Branching Statements

All of the MATLAB programs developed previously consist of a series of MATLAB statements that are executed one after another in a fixed order. Such programs are called *sequential* programs. There is no way to repeat sections of the program more than once, and there is no way to selectively execute only certain portions of the program.

We will introduce two broad categories of control statements: **branches**, which select specific sections of the code to execute, and **loops**, which cause specific sections of the code to be repeated. By using these control statements, we can control the order in which statements are executed in a program.

3.1 Branching

Branches are used to make *decisions* in your program and to run different pieces of code depending upon some ***logical condition*** that can be either ***true*** or ***false***. For example:

```
  ⋮
if   <logical condition>
     statement group A
else
     statement group B
end
  ⋮
```

If the *logical condition* is *true*, then the program executes the statements A, otherwise, the program executes the statements B. After the if-construct the program goes to the line immediately following `end`.

The value of these *logical conditions* is stored in a different MATLAB datatype called the logical datatype.

3.1.1 The Logical Data Type

The `logical` data type can have one of only two possible values: `true` or `false`. These values are produced by the two special functions `true` and `false`. They are also produced by two types of MATLAB operators: ***relational*** operators and ***logical*** operators. To create a `logical` variable, just assign a logical value to it in an assignment statement. For example, the following statement creates a logical variable `l1` containing the logical value `true`.

```
l1 = true;
```

Automatic conversion between numeric and logical datatypes:

In MATLAB, numerical and logical data can be mixed in expressions. If a logical value is used in a place where a numerical value is expected, `true` values are converted to 1 and `false` values are converted to 0, and then used as numbers. If a numerical value is used in a place where a logical value is expected, non-zero values are converted to `true` and zero values are converted to `false`, and then used as logical values.

The inbuilt function `logical()` does this conversion explicitly. For example `logical(0)` is `false` and `logical(x)`, where `x` is some non-zero number, gives `true`.

3.1.2 Relational Operators

Relational Operators are operators with two numerical or string operands that yield a `logical` result, depending on the relationship between the two operands. The general form of a relational operator is

```
a1 <op> a2
```

where `a1` and `a2` are arithmetic expressions, variables, or strings, and `<op>` is one of the following relational operators:

Summary of Relational Operators `<op>`

Operator	Interpretation
==	Equal to
∼=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Note:

- Relational operators may be used to compare a scalar value with an array.
  `logarr1 = array1 < 3`
- Relational operators may be used to compare two arrays if they have the same size.
  `logarr2 = array1 > array2`

- Relational operators may be used to compare two strings if they are of equal lengths.
  `logarr3 = 'This is string 1' == 'This is STRING 2'`
- The == symbol is a *comparison* operation that returns a logical result, while the = symbol is an *assignment* operation that assigns the value of the expression on the right of the equal sign to the variable on the left of the equal sign.
- Due to **roundoff errors** during computer calculation, two theoretically equal numbers can differ slightly. For example,

  ```
  a = 0;
  b = sin(pi);
  c = (a == b);
  ```

  The logical variable `c` should have the value of `true`, while the result of MATLAB is `false`. Because the result of `sin(pi)` MATLAB calculated is 1.2246×10^{-16} instead of exactly zero. So, instead of using *exact* 'equal to' operation, we use the following statement

  ```
  c = (abs(a - b) < 1.0E-14)
  ```

  and if `c` has the value of `true`, we consider `a` and `b` have the same values.
- All the relational operators have the same precedence.

3.1.3 Logical Array Masking

An array of logical values can be used to conduct operations on another numeric array of the *same size*. This is called **masking**. A mask is a logical array that selects only those elements of a main array that correspond to a `true` value in the mask array.

For example:

```
>> a = [1 2 3 ; 4 5 6 ; 7 8 9]
>> b = a>5
>> a(b) = sqrt(a(b))
```

The array `b` above is the masking array (it is the same size as `a`) and contains only `true` or `false` value. Thus `a(b)` is the *subarray* of `a` containing only those elements that have a `true` value in the corresponding element of `b`. Masking is helpful when one is trying to do some operations on a part of a bigger array based on some logical condition.

3.1.4 Logical Operators

On several occasions in branching, one needs to **combine** multiple logical conditions into a single condition. These is done by logical operators. Logical operators combine or operate on logical variables and yield a logical `true` or `false` result. There are five logical operators that operate on two logical variables: AND (`&` and `&&`), OR (`|` and `||`), and Exclusive-OR (`xor`), The general form of these operations is

```
l1 <op> l2
```

where `l1` and `l2` are expressions or variables.

In addition there is a **unary** operator: NOT (∼) with the general form

```
∼ l1
```

The NOT operator takes the value of `l1` and simply returns the opposite value. Thus it changes `true` to `false` and `false` to `true`.

Summary of Logical Operators `<op>`

Operator	Operation
`&`	Logical AND
`&&`	Logical AND with shortcut evaluation
`|`	Logical Inclusive OR
`||`	Logical Inclusive OR with shortcut evaluation
`xor`	Logical Exclusive OR
∼	Logical NOT

Results of Logical Operators

l1	l2	l1 & l2 l1 && l2	l1 | l2 l1 || l2	xor(l1,l2)
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

Note:
With 'shortcut evaluation' MATLAB evaluates the first expression `l1` and then decides if evaluating the second expression `l2` is going to change the result of the operation. For example, if we consider the following expression.

```
l1 = (4<5) || ( dot(1:1000,1:1000) > 0 )
```

Once MATLAB knows the value of the expression on the left of `||` is `true`, it does not matter what the result on the right is because the final result is still going to be true. MAT-

LAB simply assigns the logical constant `11` a value of `true` and moves to the next statement. Thus using shortcut evaluation can speed up your MATLAB program.

However, shortcut evaluation *cannot* be used to combine two logical arrays. Then one must use the single `|` or `&`.

Hierarchy of Operations

Logic operators are evaluated after all arithmetic operations and all relational operators have been evaluated.

1. All arithmetic operators are evaluated first in the order previously described.
2. All relational operators are evaluated, working from left to right.
3. All $\sim$ operators are evaluated.
4. All `&` and `&&` operators are evaluated, working from left to right.
5. All `|`, `||`, and `xor` operators are evaluated, working from left to right.

3.2 The `if` branch

The simplest form of an `if-end` statement allows one to decide whether or not to execute certain block of code. The syntax of an `if-end` is:

`if` *logical expression*
  statements A
`end`

If *logical expression* is `true`, the program executes the statements A. Otherwise, the program skips the statements in between the `if-end` construct and goes to the line immediately following `end`. A short form of this construct can also be used:

`if` *logical expression* , statement A , `end`

One may choose to execute a different section of code if the *logical expression* is not true. This can be done with the `if-else-end` construct, which has the syntax:

`if` *logical expression*
  statement group A
`else`
  statement group B
`end`

If the *logical expression* is `true`, then statement group A is executed. If the *logical expression* is `false`, then statement group B is executed. A short form of this construct can also be used:

`if` *logical expression* , statement A , `else` statement B , `end`

Sometimes, one may have to check multiple conditions one after the other before deciding what section of code to execute. This can be done by adding the `elseif` clause to the above `if-end` syntax. The general form of the `if-elseif-end` construct has the following syntax:

`if` *logical expression 1*
  statement group A
`elseif` *logical expression 2*
  statement group B
`elseif` *logical expression 3*
  statement group C
⋮
`end`

Here, if *logical expression 1* is `true`, then **only** statement group A is executed.
otherwise if *logical expression 2* is `true`, then **only** statement group B is executed.

otherwise if *logical expression 3* is `true`, then **only** statement group C is executed.

Note:
If both the *logical expressions 1 and 2* are `true`, then **only** statement group A is executed.
If **none** of the logical expressions is `true`, then **none** of the statements within the `if-elseif-end` structure is executed.

Another variation of this construct includes an `else` clause towards the end resulting in an `if-elseif-else-end` construct:

`if` *logical expression 1*
  statement group A
`elseif` *logical expression 2*
  statement group B
`elseif` *logical expression 3*
  statement group C
⋮
`else`
  statement group D
`end`

Note:
In this case, if **none** of the logical expressions 1, 2 and 3 is `true`, then the statement group D is executed.

3.2.1 The Nested `if` Statement

When one `if` branch is completely contained within the statement group of another outer `if` branch, the process is called **nesting**. In general it takes the form:

`if` *logical expression 1*
  statement group A
  `if` *logical expression 2*
    statement group B
  `end`
  statement group C
`end`

In this case, statement group B is executed **only if** first, the *logical expression 1* is `true` and then if *logical expression 2* is `true`.

3.3 The `switch` statement

The `switch` construct is used when you want to **match** the value of a variable or expression with a set of different values and then choose to execute a particular block of code. The variable or expression to be matched is called the *switch expression* and it is matched with different *case expressions*. Both these types of expressions can be integer, character-string, or logical expressions.

The general form of a `switch` construct is:

```
switch (switch expression)
case case expression 1,
   statement group A
case case expression 2,
   statement group B
...
otherwise,
   statement group N
end
```

If the value of *switch expression* is equal to *case expression 1*, then 'statement group A' will be executed, and the program will jump to the first statement following the end of the `switch` construct. Similarly, if the value of *switch expression* is equal to *case expression 2*, then 'statement group B' will be executed, and the program will jump to the first statement following the end of the `switch` construct. The same idea applies for any other cases in the construct.

The `otherwise` code block is optional. If it is present, it will be executed whenever the value of *switch expression* is outside the range of all the case selectors. If it is not present and the value of *switch expression* is outside the range of all the case selectors, then none of the code blocks will be executed.

If many values of the *switch expression* should cause the same code to execute, all of those values may be included in a single block by enclosing them in **curly** brackets. In the following example, if the *switch expression* matches any of the three *case expressions* in the list, then 'statement group a' will be executed.

```
switch (switch expression)
case {case expression 1, case expression 2, case expression 3},
   statement group A
...
otherwise,
   statement group N
end
```

At most one code block can be executed. After a code block is executed, execution skips to the first executable statement after the `end` statement. If the *switch expression* matches more than one *case expression*, only the **first** one of them will be executed.

3.4 MATLAB Debugger

The debugger is a useful tool integrated with the MATLAB editor that helps you find errors in your code and follow the execution of your code line by line.

- **Breakpoints:**
  You can stop the execution of your program at a certain point in your code by setting a **breakpoint**. Once MATLAB reaches that point in the code it 'freezes' execution and returns the control to the user. At this point you can check all your variables and run commands as you normally would. To proceed from a breakpoint you may press F10 to follow your code line by line or press F5 to continue execution till the next breakpoint (if any).

- **Conditional Breakpoints:**
  Conditional breakpoints can be set when you wish to stop execution of your code at a point depending upon certain 'logical' condition being true. For example, stop only if a variable has a negative value or only for a certain value of the loop variable.

- **Error Breakpoints:**
  Execution stops if any errors or warnings are encountered.

- **Check code with M-lint:**
  M-lint checks for errors, obsolete features, unused variables etc.

- **Profiler:**
  Shows the details of the computational time taken to execute a program. Thus one can see which parts of the program take up more time and try to make their that part of the calculation more *efficient*.

Chapter 4

Loops

As your programs start to get more complicated, you will need to follow some Program Design techniques before you can begin writing the code.

4.1 Top-Down Design Techniques

Top-down design is the process of starting with a large task and breaking it down into smaller, more easily understandable pieces (sub-tasks) which perform a portion of the desired task. Each sub-task may in turn be subdivided into smaller sub-tasks if necessary. Once the program is divided into small pieces, each piece can be coded and tested independently. The sub-tasks will be combined into a complete task after each of the sub-tasks has been verified to work properly by itself.

The concept of top-town design is the basis of program design process. The details of the process are shown in Figure 3.1.

Program design process

1. *Clearly state the problem that you are trying to solve.*
2. *Define the inputs required by the program and the outputs to be produced by the program.*
3. *Design the algorithm that you intend to implement in the program.*
   An **algorithm** is a step-by-step procedure for finding the solution to a problem. At this stage, the top-down design techniques are used.
4. *Turn the algorithm into MATLAB statements.*
5. *Test the resulting MATLAB program.*

Algorithms are usually described using a **pseudo-code** (derived from pseudo meaning false or stand in, and code for program) that uses constructs such as branching and loops similar to a programming language but does not have a set syntax. We will use a mixture of MATLAB and English as pseudo-code in this class.

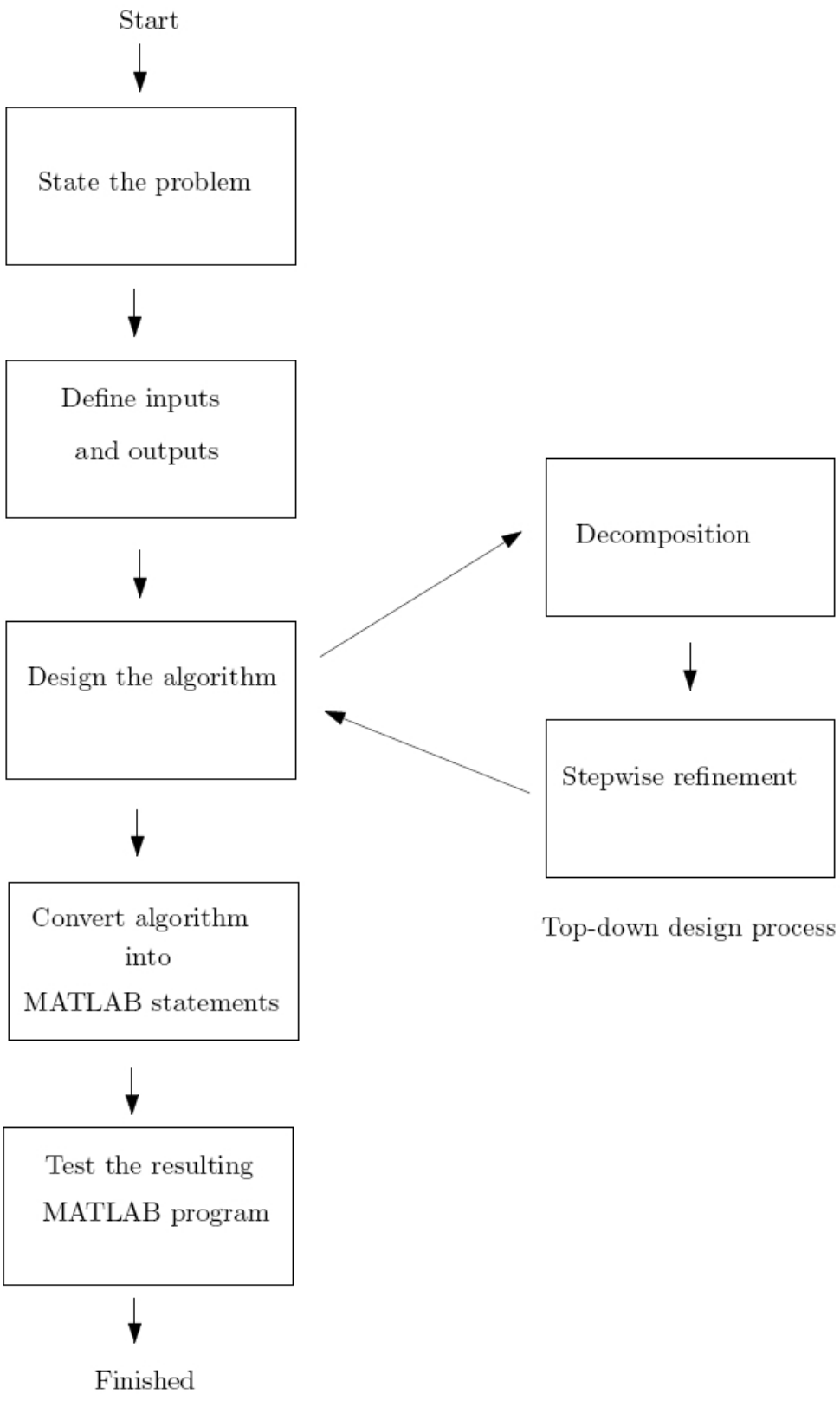

Figure 4.1: The program design process

4.2 Loops

One of the strongest attributes of a computer is its ability to do fast **repetitive** operations on a set of data. As programmers, we use this feature through **loops** when we want to repeat certain parts of our program over and over again. For instance, say we want to calculate the sum of all the elements of an array `A`. The 'brute force' way of doing this would be:

```
>> sumA = A(1) + A(2) + A(3) + . . . + A(N)
```

where `N` is the total number of elements in the array. Another way to write the same is:

```
>> sumA = 0
>> sumA = sumA + A(1)
>> sumA = sumA + A(2)
>> sumA = sumA + A(3)

   ⋮              ⋮

>> sumA = sumA + A(N)
```

Clearly, this will be very inefficient for large arrays. However, there is a **repetitive** pattern in the above expressions which can be expressed in the following formula:

$$\texttt{sumA} = \sum_{i=1}^{N} \texttt{A}(i)$$

Having identified the pattern, the formula can be directly coded up in MATLAB as a `for` loop as:

```
sumA = 0
for ii = 1 : N
      sumA = sumA + A(ii)
end
```

The way MATLAB implements this loop is by **repeatedly** executing the statement between the `for-end` construct for **each** value of `ii` from `1` to `N`. In general, as a programmer, if you find yourself doing repetitive operations then you should immediately try to write down an expression that defines the pattern (as in the above equation) and use a loop to implement it.

In MATLAB there are two basic forms of loop constructs: **for loops** and **while loops**. The major difference between these two types of loops is in how the repetition is controlled. The code in a `for` loop is repeated a specified number of times, and the number of repetitions is known before the loops starts. By contrast, the code in a `while` loop is repeated an indefinite number of times until some user-specified condition is satisfied.

4.3 The `for` Loop

The general form of a `for` loop is

`for` *index* = *expression*
  statement group (*body* of the loop)
`end`

The *expression* usually takes the form of a vector in shortcut notation `first:incr:last`. The *index* variable is assigned values from the loop *expression* incrementally for every *pass* of the loop. The number of times the loop will be executed can be computed using the following equation: `floor((last-first)/incr)+1`. If the value is negative, the loop is not executed.

Example: Calculate the summation of `1 + 2 + ... + 100`.

```
sum = 0;
for ii=1:100
  sum = sum + ii;
end
```

Good Programming Practice

1. Indent the bodies of loops.
2. Do **not** modify the loop index `ii` within the body of a `for` loop.

4.3.1 The general form of the `for` Loop

In `for` loops, the loop *expression* can take other forms also. In general, the loop *expression* can be any matrix. Then, MATLAB assigns one column of the matrix to the *index* variable and executes the statements within the body of the `for-end` loop, successively for as many columns there are in the matrix.

Example:

```
for ii = [5 9 7]
  ii
end
```

Example:

```
for ii = [1 2 3; 4 5 6]
  ii
end
```

The `for` loop construct functions as follows:

1. At the beginning of the loop, MATLAB generates the control expression.
2. The first time through the loop, the program assigns the first column of the expression to the loop variable `index`, and the program executes the statements within the body of the loop.
3. After the statements in the body of the loop have been executed, the program assigns the next column of the expression to the loop variable `index`, and the program executes the statements within the body of the loop again.
4. The above step 3 is repeated over an over as long as there are additional columns in the control expression.

Note:

- The index of the `for` loop must be a variable.
- If the *expression* is a scalar, then the loop will be executed one time, with the index containing the value of the scalar.
- If the *expression* is a row vector, then each time through the loop, the index will contain the next value in the vector.
- If the *expression* is a matrix, then each time through the loop, the index will contain the next column in the matrix.
- Upon completion of the loop, the index contains the last value used.

4.4 The `while` Loop

The general form of a `while` loop is

`while` *expression*
  statement group
`end`

The controlling expression produces a logical value. If the *expression* is `true`, the statement group will be executed, and then control will return to the `while` statement. If the *expression* is still `true`, the statements will be executed again. The process will be repeated until the *expression* becomes `false`. When control returns to the `while` statement and the *expression* is `false`, the program will execute the first statement after the `end`.

If the *expression* is always `true` (for example, we made an mistake in the *expression*), the loop becomes an infinite loop and we need to use the **Ctrl-C** key to abort it.

Example: Calculate the summation of `1 + 2 + ... + 100`.

```
sum = 0;
current = 1;
while current <= 100
   sum = sum + current;
   current = current + 1;
end
```

4.5 Simple Applications

1. Finding **Statistical** Average and Standard Deviation of some numbers:

```
numbers = input('Enter an array of numbers: ')
n = length(numbers);
sumnum = 0;
sqrsum = 0;
for ii = 1:n
    sumnum = sumnum + numbers(ii);
    sqrsum = sqrsum + numbers(ii)^2;
end
avg = sumnum/n;
stddev = sqrt((n*sqrsum-sumnum^2)/(n*(n-1)));
fprintf('The mean is %f and the standard deviation is %f. \n', avg, stddev);
```

2. **Searching** for a number in a given array:

```
numbers = input('Enter an array of numbers: ')
srch = input('Enter the number to search: ');
loc = [];
for ii = 1:length(numbers)
    if numbers(ii) == srch
        loc = [loc ii];
    end
end
if length(loc) > 0
    fprintf('The number %d occurs at the indices: ', srch);
    fprintf(' %d ', loc);
    fprintf('in the array. \n');
else
    fprintf('The number %d was not found in the array: ', srch);
end
```

3. Find **minimum** or **maximum** number in an array:

```
numbers = input('Enter an array of numbers: ')
minnum = Inf;
maxnum = 0;
minloc = [];
maxloc = [];
for ii = 1:length(numbers)

    if numbers(ii) < minnum
        minnum = numbers(ii);
        minloc = [ii];
    elseif numbers(ii) == minnum
        minloc = [minloc ii];
    end

    if numbers(ii) > maxnum
        maxnum = numbers(ii);
        maxloc = [ii];
    elseif numbers(ii) == maxnum
        maxloc = [maxloc ii];
    end

end
fprintf('The minimum number is: %d \n' , minnum);
fprintf('It was found at the following locations in the array:');
fprintf(' %d ', minloc);
fprintf('\n');
fprintf('The maximum number is: %d \n' , maxnum);
fprintf('It was found at the following locations in the array:');
fprintf(' %d ', maxloc);
fprintf('\n');
```

4.6 Timing, Preallocation and Vectorization of Loops

The commands `tic` and `toc` can be used to calculate the the time taken for a certain MATLAB code to execute.

```
clear all; clc;
tic;
for ii = 1:10000
    sqr(ii) = ii^2;          % The size of sqr is GROWING in the loop as ii grows
end                          % MATLAB has to allocate & deallocate memory
toc                          % and transfer data which takes a lot of time.
```

Preallocation of arrays before the loop begins can speed up the execution.

```
clear all; clc;
% With Preallocation
tic;
sqr = zeros(1,10000);        % Here we PREALLOCATE all the memory needed by sqr
for ii = 1:10000
    sqr(ii) = ii^2;
end
toc
```

Alternatively, you can also use **Vectorization** i.e. use the *array* operators (`.* ./ .^` etc.) to speed up the execution even more.

```
clear all; clc;
% Vectorization
tic;
ii = 1:10000;
sqr = ii.^2;                 % Vectorization as an alternative to loops
toc
```

Note that the `for` loop has been replaced entirely by the *array* operations. MATLAB internally implements the loop for you when you use array operations. It does so using *vectorized* memory operations which are faster.

4.7 The `break` and `continue` Statements

The `break` and `continue` statements can be used to control the operation of `while` loops and `for` loops. The `break` statement terminates the execution of a loop and passes control to the next statement after the end of the loop, while the `continue` statement terminates the current pass through the loop and returns control to the top of the loop.

Examples:

```
for ii = 1:5
   if ii == 3
      break;
   end
   fprintf('ii = %d \n', ii);
end
disp('End of loop!');


for ii = 1:5
   if ii == 3
      continue;
   end
   fprintf('ii = %d \n', ii);
end
disp('End of loop!');
```

4.8 Nested Loops

It is also possible to have one loop be completely inside another loop. If one loop is completely inside another one, the two loops are called **nested loops**. Nested loops are commonly used for doing computations on Matrices.

Example: Ask user for a row vector `a`. Calculate `A = a' * a`

```
clear all;
a = input('Enter a row vector: ')
n = length(a);
for ii = 1:n
    fprintf('ii = %d \n', ii);
    for jj = 1:n
        fprintf('     jj = %d \n', jj);
        A(ii,jj) = a(ii)*a(jj);
    end
end
A
```

Gives the output:

```
a =
     2     5     7
ii = 1
     jj = 1
     jj = 2
     jj = 3
ii = 2
     jj = 1
     jj = 2
     jj = 3
ii = 3
     jj = 1
     jj = 2
     jj = 3

A =
     4    10    14
    10    25    35
    14    35    49
```

Note:

- MATLAB associates each `end` statement with the *latest open* `for` loop. You **cannot** have an `end` for the outer loop before the `end` for the inner loop.
- When `for` loops are nested, they should have **independent loop index** variables.
- When a `break` or `continue` statement appears inside a set of nested loops, then it refers to the *latest open* loops containing it.

Example:

```
clear all;
a = input('Enter a row vector: ')
n = length(a);
for ii = 1:n
    fprintf('ii = %d \n', ii);
    if (ii == 3)
        fprintf('Outer Loop ii == 3 pass SKIPPED. \n');
        continue;
    end
    for jj = 1:n
        fprintf('     jj = %d \n', jj);
        if (jj == 3)
            fprintf('Inner Loop BROKEN at jj == 3. \n');
            break;
        end
        A(ii,jj) = a(ii)*a(jj);
    end
end
A
```

Chapter 5

More Plotting and Graphics

Recall the basic usage of `plot()` command. An example of a complete plot is shown in Figure 5.1, and the statements to produce that plot are shown below.

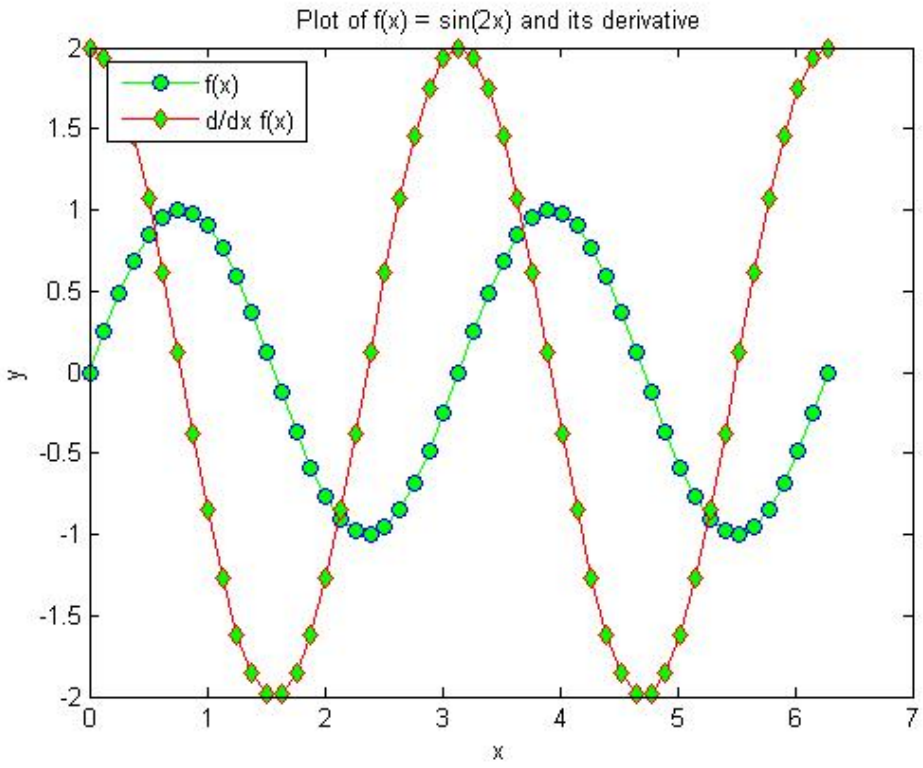

Figure 5.1: A complete plot with title, axis labels, legend, grid, and multiple line styles.

```
x = 0:pi/25:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x, y1, 'go-', 'MarkerSize', 6.0, 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'g');
hold on;
```

```
plot(x, y2, 'rd-', 'MarkerSize', 6.0, 'MarkerEdgeColor', 'r', 'MarkerFaceColor', 'g');
title('Plot of f(x) = sin(2x) and its derivative');
xlabel('x');
ylabel('y');
legend('f(x)', 'd/dx f(x)', 'Location', 'NorthWest');
```

5.1 Additional Types of Two-dimensional Plots

Some of the additional two-dimensional plotting functions are listed here.

1. `bar(x, y)` creates a *vertical* bar plot, with the values in `x` used to label each bar and the values in `y` used to determine the height of the bar.
2. `barh(x, y)` creates a *horizontal* bar plot, with the values in `x` used to label each bar and the values in `y` used to determine the horizontal of the bar.
3. `compass(x, y)` creates a polar plot, with an arrow drawn from the origin to the location of each (`x, y`) point.
4. `pie(x)` creates a pie plot. This function determines the percentage of the total pie corresponding to each value of `x` and plots pie slices of that size.
5. `stairs(x, y)` creates a stair plot, with each stair step centered on an (`x, y`) point.
6. `stem(x, y)` creates a stem plot, with a marker at each (`x, y`) point and a stem drawn vertically from that point to the x axis.
7. `errorbar(X,Y,L,U)` plots `X` versus `Y` with error bars. For each point defined by `(X(i),Y(i))`, the vectors `L`, and `U` represents the distance `L(i)` below and `U(i)` above the point for the error bar.

5.1.1 Other Useful Plotting Functions

1. `ezplot('functionname',[xmin xmax], figure)`
2. `fplot('functionname',[xmin xmax])`

Example:

```
ezplot('sin(x)/x',[-4*pi 4*pi])
```

5.1.2 Logarithmic Plots

It is possible to plot data on logarithmic scales as well as linear scales. There are four possible combinations of linear and logarithmic scales on the x and y axes, and each combination is produced by a separate function.

1. The `plot` function plots both x and y data on linear axes.
2. The `semilogx` function plots x data on logarithmic axes and y data on linear axes.
3. The `semilogy` function plots x data on linear axes and y data on logarithmic axes.
4. The `loglog` function plots x and y data on logarithmic axes.

5.1.3 Subplots

It is possible to place more than one set of axes on a single figure, creating multiple **subplots**. Subplots are created with a `subplot` command of the form

```
subplot(m,n,p)
```

This command divides the current figure into $m \times n$ equal-sized regions, arranged in m rows and n columns, and creates a set of axes at position p to receive all current plotting commands. The subplots are numbered from left to right and from top to bottom. For example, the command `subplot(2, 3, 4)` would divide the current figure into six regions arranged in two rows and three columns, and create an axis in position 4 (the lower left one) to accept new plot data.

Example: Figure 5.2 is produced by the following code:

```
x = 0:pi/30:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);

figure(1)

subplot(2,2,1)
plot(x, y1, 'k-', 'LineWidth', 2.0);
xlabel('x');
ylabel('f(x)');

subplot(2,2,2)
semilogx(x, y1, 'b-', 'LineWidth', 3.0)
xlabel('x');
ylabel('f(x)');

subplot(2,2,3)
```

```
plot(x, y2, 'r-o', 'LineWidth', 2.0);
xlabel('x');
ylabel('d/dx f(x)');
axis([0 6 -3 3]);

subplot(2,2,4)
plot(x, y2, 'kd', 'MarkerSize', 6.0, 'MarkerEdgeColor', 'r', 'MarkerFaceColor', 'g');
xlabel('x');
ylabel('d/dx f(x)');
axis([0 6 -3 3]);
```

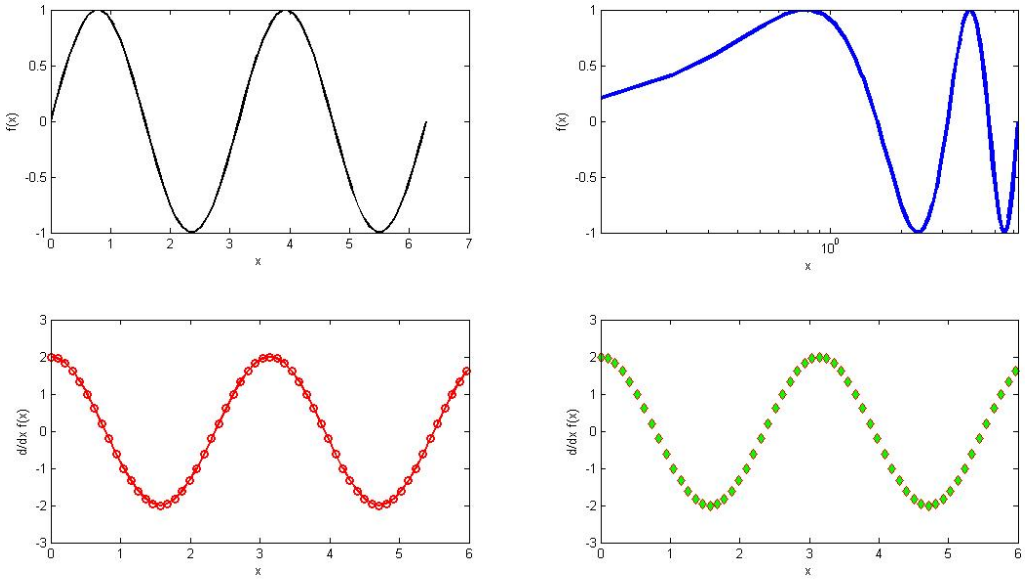

Figure 5.2: A figure containing four subplots.

5.1.4 Creating Multiple Figure Windows

Matlab can create multiple Figure Windows, with different data displayed in each window. Window is identified by a *figure number*, which is a small positive integer. The first Figure Window is Figure 1, the second is Figure 2, etc. One of the Figure Windows will be the **current figure**, and all new plotting commands will be displayed in that window.

The current figure is selected with the `figure` function. This function takes the form `figure(n)`, where `n` is a figure number. When this command is executed, Figure `n` becomes the current figure and is used for all plotting commands. The figure is automatically created if it does not already exist. The current figure may also be selected by clicking on it with the mouse.

5.1.5 Exporting a Plot as a Graphical Image

The `print` command can be used to save a plot as a graphical image by specifying appropriate options and a file name.

```
print -f<handle> -<options> <filename>
```

where `-f<handle>` specifies Graphics handle of figure to print, `-<options>` specifies the format of the output image, and `<filename>` specifies the name of the output image. For example,

```
print -f1 -djpeg myplot.jpg
```

creates a JPEG image of the figure 1 and store it in a file called `myplot.jpg`. Other options allow image files to be created in other formats. Some of the important image file formats are given in the following table:

`print` Options to Create Graphics Files

Option	Description
`dps`	PostScript for black and white printers
`dpsc`	PostScript for color printers
`deps`	Encapsulated PostScript
`depsc`	Encapsulated Color PostScript
`djpeg`	JPEG image
`dtiff`	Compressed TIFF image
`dpng`	Portable Network Graphic color image

5.2 Three-dimensional Plots

Three-dimensional plots are needed when we want to display plots of functions involving *three* variables, x, y and z.

5.2.1 `plot3` function

This function plots a 3-D *curve* by connecting (x, y, z) triplets with line segments (similar to the `plot` command which connects (x, y) pairs). General format:

```
plot3(x, y, z)}
```

where `x`, `y`, and `z` are vectors of equal length containing the locations of data points to plot. Figure 5.3 is produced by the following code:

```
z = 0:pi/50:10*pi;
plot3(sin(z),cos(z),z);
```

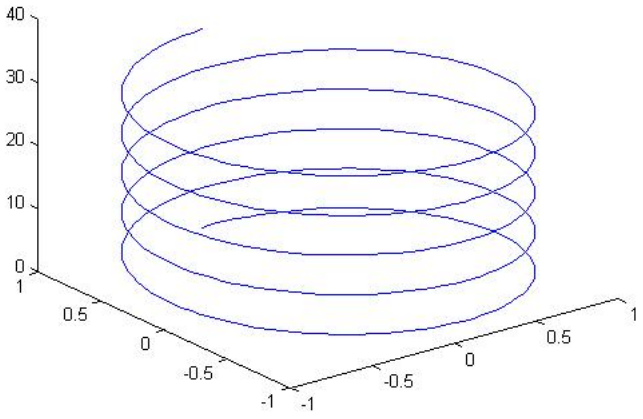

Figure 5.3: A three-dimensional line plot.

5.2.2 The `meshgrid`, `mesh` and `surf` commands

For plotting a function of two variables `z(x,y)` we use the `mesh` and `surf` commands. Figure 5.4 and Figure 5.5 are produced by the `mesh` and `surf` functions in the following code.

```
[x, y] = meshgrid(-4:0.2:4);
z = exp(-0.5*(x.^2 + y.^2));
figure(1)
mesh(x, y, z);
figure(2)
surf(x, y, z);
```

Note: The `meshgrid` command is only used to generate the `x` and `y` arrays that represent the 'grid' of (x, y) pairs. The function $z(x, y)$ is then calculated upon this grid of values and plotted using `mesh` and `surf` commands.

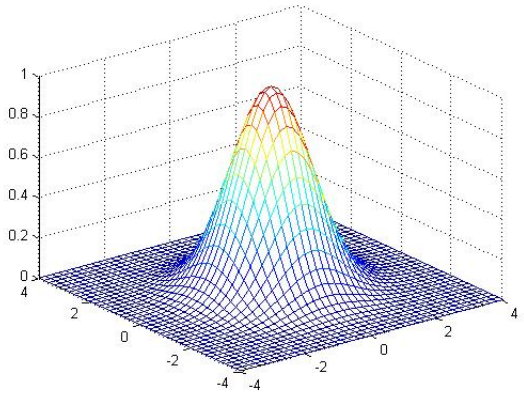

Figure 5.4: A three-dimensional mesh plot.

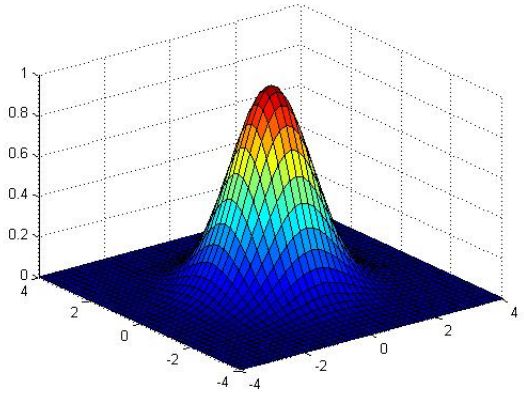

Figure 5.5: A three-dimensional surf plot.

5.2.3 The Contour functions

Another way to visualize a function of two variables is to use contour plots. MATLAB has can generate contour plots with functions such as: `contour()`, `contour3()` and `contourf()`. A sample code and its output for the three functions is shown below.

```
[x, y, z] = peaks(50);
subplot(2,2,1);
surf(x, y, z); title('surf')
subplot(2,2,2);
contour(x, y, z); title('contour')
subplot(2,2,3);
contour3(x, y, z); title('contour3')
subplot(2,2,4);
contourf(x, y, z); title('contourf')
```

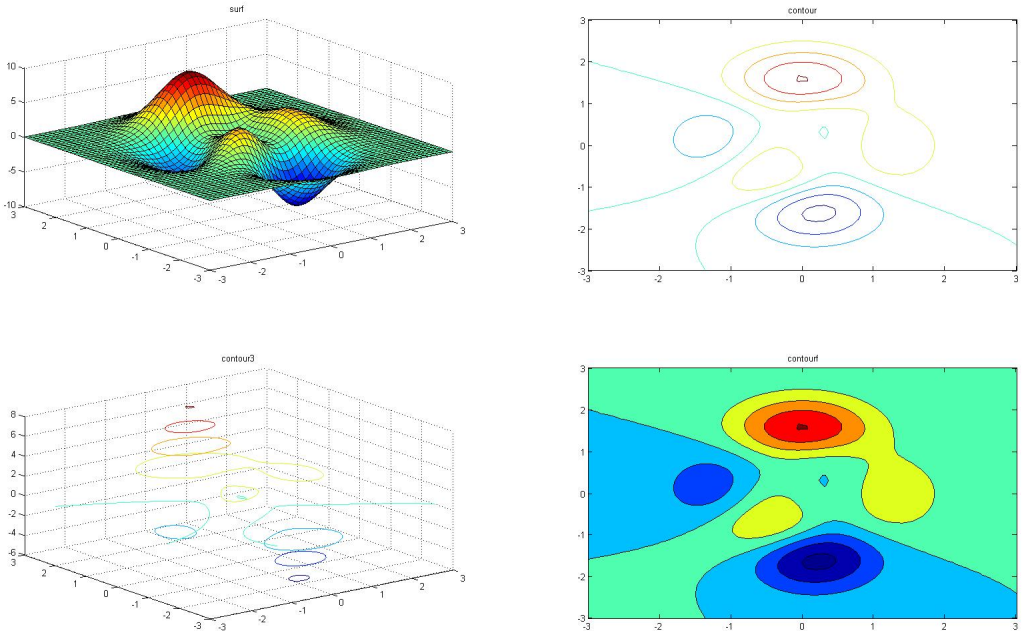

Figure 5.6: MATLAB contour plots.

5.2.4 Generating Animations of Plots

Another example of a repetitive process is encountered when one is trying to generate an animation (movie) of plots. Generating a **movie** can be done first creating a plot and then *over-writing* it with another (slightly different) plot and doing this **repeatedly** over and over again.

```
delay = pi/50;
t = 0:delay:2*pi;
y1 = sin(2*t);
y2 = 2*cos(2*t);
for ii = 1:length(t)
    hold off
    plot(t(1:ii), y1(1:ii), 'go-', 'MarkerSize', 6.0, 'MarkerEdgeColor', ...
         'b', 'MarkerFaceColor', 'g');
    hold on;
    plot(t(1:ii), y2(1:ii), 'rd-', 'MarkerSize', 6.0, 'MarkerEdgeColor', ...
         'r', 'MarkerFaceColor', 'g');
    axis([0 2*pi -2 2]);
    text(5.0,1.8,['t = ' num2str(t(ii))]);
    pause(delay);
end
```

The `axis` command is needed to keep the y-axis constant during the animation. The `text` command displays the time and the `pause` command allows the user to see each plot for the specified `delay` (in seconds) before it is changed.

3-D plots can also be animated. For example:

```
[x y] = meshgrid(-pi:pi/10:pi,-pi:pi/10:pi);
figure(1); clf;
for t = 1:20                            % Loop over time limits for animation
    z = sin(t/20*2*pi)*cos(x).*sin(y);  % create the z values
    surf(x, y, z);                      % plot the surface at this instant
    axis([-pi pi -pi pi -2 2]);         % set axis limits
    text(2,-2,2,['t=' num2str(t)]);     % output text
    pause(0.1);                         % pause for users to see the image
end
```

Chapter 6

User Defined Functions, Recursion

Why do we need user-defined functions? Well-designed functions enormously reduce the effort required on a large programming project. Their benefits include:

1. **Independent testing of sub-tasks**.
2. **Reusable code**.
3. **Isolation from unintended side effects**.

6.1 Introduction to Matlab Functions

The general form of a Matlab function is

```
function [outarg1, outarg2, ...] = fname(inarg1, inarg2, ...)
% H1 comment line
% Other comment lines

(Executable code)
...
(return)
(end)
```

For example, suppose in a Matlab program, many times you need convert Spherical coordinates, (r, θ, ϕ), to Cartesian coordinates, (x, y, z). Your code may look like this:

```
...
% converts from (r1, theta1, phi1) to (x1, y1, z1)
z1 = r1*cos(phi1);
x1 = r1*sin(phi1)*cos(theta1);
y1 = r1*sin(phi1)*sin(theta1);
...
% converts from (r2, theta2, phi2) to (x2, y2, z2)
z2 = r2*cos(phi2);
x2 = r2*sin(phi2)*cos(theta2);
y2 = r2*sin(phi2)*sin(theta2);
...
% converts from (r3, theta3, phi3) to (x3, y3, z3)
z3 = r3*cos(phi3);
x3 = r3*sin(phi3)*cos(theta3);
y3 = r3*sin(phi3)*sin(theta3);
...
```

Alternately if a user defined function, which converts from Spherical coordinates to Cartesian coordinates, had been used, your code may look like this:

```
...
% converts from (r1, theta1, phi1) to (x1, y1, z1)
[x1, y1, z1] = Spherical2Cartesian(r1, theta1, phi1);
...
% converts from (r2, theta2, phi2) to (x2, y2, z2)
[x2, y2, z2] = Spherical2Cartesian(r2, theta2, phi2);
...
% converts from (r3, theta3, phi3) to (x3, y3, z3)
[x3, y3, z3] = Spherical2Cartesian(r3, theta3, phi3);
...
```

where function `Spherical2Cartesian` is defined in the Matlab script file `Spherical2Cartesian.m`.

```
function [x, y, z] = Spherical2Cartesian(r, theta, phi)
% converts from (r, theta, phi) to (x, y, z).

z = r*cos(phi);
x = r*sin(phi)*cos(theta);
y = r*sin(phi)*sin(theta);

return;
```

Notes

1. The `function` statement specifies the name of the function and the input and output argument lists. The input argument list appears in parentheses after the function name, and the output argument list appears in brackets to the left of the equal sign. If there is only one output argument, the brackets can be dropped.

2. Matlab function should be placed in a file with the same name (including capitalization) as the function, and the file extent `.m`. For example, if a function is named `Myfun`, then that function should be placed in a file named `Myfun.m`.

3. The input argument list is a list of names representing values that will be passed from the caller to the function. These names are called **dummy arguments**. They are just placeholders for actual values that are passed from the caller when the function is invoked. Similarly, the output argument list contains a list of dummy arguments that are placeholders for the values returned to the caller when a function finishes executing.

4. A function is invoked by naming it in an expression together with a list of **actual arguments**. A function may be invoked by typing its name directly in the Command Window or by including it in a script file or another function. The name in the calling program must *exactly match* the function name (including capitalization). When the function is invoked, the value of the first actual argument is used in place of the first dummy argument, and so forth for each other actual argument/dummy argument pair.

5. Execution begins at the top of the function and ends when either a `return` statement, an `end` statement, or the end of the function is reached. The `return` statement or the `end` statement is not actually required in most functions. **The only information returned from the function is contained in the output arguments**. So, each item in the output argument list must appear on the left side of at least one assignment statement in the function. When the function returns, the values stored in the output argument list are returned to the caller.

6. Matlab function is a special type of M-file that runs in its own independent workspace. For example, the variable defined in the main program, i.e. `x`, has nothing to do with the variable in a function although they may have the same name `x`. In other words, the change of the value of `x` in the function does not change the value of `x` in the main program or vice versa.

7. The initial comment lines in a function serve a special purpose. The first comment line after the function statement is called the **H1 comment line**. It should always contain a one-line summary of the purpose of the function. The special significance of this line is that it is searched and displayed by the `lookfor` command. The comment lines from the H1 line until the first blank line or the first executable statement are displayed by the `help` command. They should contain a brief summary of how to use the function.

Examples:

Single input and single output function

```
function volume = spherevol(r)
% This function calculates the volume of spheres
volume = 4/3*pi*r.*r.*r;
```

Multiple inputs and single output function

```
function volume = cylindervol(r, h)
% This function calculates the volume of cylinders
volume = pi*r.*r.*h;
```

Single input and multiple outputs function

```
function [sarea, volume] = spheresurvol(r)
% This function calculates the volume and surface area of spheres
volume = 4/3*pi*r.*r.*r;
sarea = 4*pi*r.*r;
```

Multiple inputs and multiple outputs function

```
function [sarea, volume] = cylindersurvol(r, h)
% This function calculates the volume and surface area of cylinders
sarea = 2.0*pi*r.*(h + r);
volume = pi*r.*r.*h;
```

Searching

```
function [locations] = mysearch(arr, num2search)
% This function searches for a given number (num2search) in a given array (arr).
% It returns an array of the indices [locations] where the number is found.

locations = [];
tol = 1e-14;
n = length(arr);
for ii = 1:n
  if abs(arr(ii)-num2search)<tol
    locations = [locations ii];
  end
end
```

Sorting

```
function sortedarr = mysort(inparr)
% This function sorts a given array of numbers (inparr) using selection sort

n = length(inparr);

for ii = 1:n-1
%    inparr
%    pause
    min = inparr(ii);
    loc = ii;
    for jj = ii + 1:n
        if inparr(jj)<min
            loc = jj;
            min = inparr(jj);
        end
    end
    if loc ~= ii  % Need to swap inparr(loc) and inparr(ii)
        inparr(loc) = inparr(ii);
        inparr(ii) = min;
    end
end

sortedarr = inparr;
```

6.2 Variable Passing in Matlab: The Pass-by-Value Scheme

Matlab programs communicate with their functions using a **pass-by-value** scheme. When a function call occurs, Matlab makes a *copy* of the actual arguments and passes them to the function. This copying is highly significant, because it means that even if the function modifies the input arguments, it won't affect the original data in the caller. This feature helps to prevent unintended side effects, in which an error in the function might unintentionally modify variables in the calling program.

This behavior is illustrated in the function shown below. This function has two input arguments: `a` and `b`. During its calculation, it modifies both input arguments.

```
function out = sample(a, b)
fprintf('In sample:  a = %f, b= %f %f \n', a, b);
a = b(1) + 2*a;
b = a.*b;
out = a + b(1);
fprintf('In sample:  a = %f, b= %f %f \n', a, b);
```

A simple test program to call this function is shown below.

```
a = 2; b = [6 4];
fprintf('Before sample:  a = %f, b= %f %f \n', a, b);
out = sample(a, b);
fprintf('After sample:  a = %f, b= %f %f \n', a, b);
fprintf('After sample:  out = %f \n', out);
```

When this program is executed, the results are:

```
Before sample:  a = 2.000000, b= 6.000000 4.000000
In sample:  a = 2.000000, b= 6.000000 4.000000
In sample:  a = 10.000000, b= 60.000000 40.000000
After sample:  a = 2.000000, b= 6.000000 4.000000
After sample:  out = 70.000000
```

Note that `a` and `b` were both changed inside the function `sample`, but those changes had *no effect on the values in the calling program.*

6.3 Optional Arguments

- `nargin`
  Returns the actual numbers of input arguments that the function was called with.
- `nargout`
  Returns the actual numbers of ouput arguments that the function was called with.
- `msg = nargchk(minarg, maxarg, nargs) % nargs = nargin OR nargout`
  Generates a standard message if the function was called with incorrect number of input OR output arguments. If there is no error then `msg=''` is empty
- `error(msg), warning(msg)`
  Displays the error or warning message `msg`. If `msg` is empty then no action is taken.

Aside: Global and Persistent variables
Global variables are declared using the keyword `global` and can be used within all functions that declare them as global. They are present in a declaring function's workspace but they are not lost when the function exits after completion.
Persistent variables are local to some function that declares them with the keyword `persistent`. The difference between persistent variables and normal function variables is that when a function is called again and again, persistent variables preserve their value between calls.

6.4 Function of functions

Function functions are useful when you need to specify the name of a MATLAB function or a user defined function as an input argument.

- `feval('functionname', x1, ... xN)`
  Evaluates the specified function *`functionname`* with inputs *`x1, ... xN`*.
- `eval('string')`
  Executes the given *`'string'`* as a MATLAB statement.
- Examples `fzero, quad, fminbnd`

6.5 Recursive Functions

An important class of functions are Recursive functions. A function is said to be recursive **if it calls itself** in its own definition. Recursion is useful for computing the result of a function which can be expressed in terms of an integer `(n)` number of **repetitive** operations. In such cases, one can express the function itself as recursive formula. For example, the sum of first `n` integers can be written as:

$$\begin{aligned} S(n) = & \quad \underline{1+2+3+...+(n-1)} \quad +n \\ S(n) = & \quad\quad S(n-1) \quad\quad\quad +n \end{aligned}$$

The first equation shows a *non*-recursive way of calculating the sum of first `(n)` integers. This equation can be implemented using the familiar loops.
The second equation defines a recursive formula for calculating the sum. The MATLAB code corresponding to the second **recursive** equation above:

```
function [outsum] = sumrec(n)
% sumnonrec : calculates the sum of the first n integers using recursion
if n<1
    error('Error: n must be positive \n');
elseif n==1
    outsum = 1;
else
    outsum = sumrec(n-1) + n;
end
```

Important points about Recursion

1. Any recursive function **calls itself** in its own definition.
   In the above function `sumrec()` calls itself with a an input that is **one less** `(n-1)` than the given input `(n)`. Thus, when a user calls this function with (say) `sumrec(10)`, the function calls itself to find `sumrec(9)`, which in-turn calls `sumrec(8)` and so on:

   ```
   sumrec(10):    outsum = sumrec(9) + 10;
   sumrec (9):    outsum = sumrec(8) + 9;
   ⋮                ⋮          ⋮
   sumrec (2):    outsum = sumrec(1) + 2;
   sumrec (1):    outsum = 1;
   ```

   In the end, when `sumrec(2)` calls `sumrec(1)`, recursion stops and the value of `sumrec(1)` = 1 is returned. Then `sumrec(2)` adds this value to 2 and returns 3 and so on back upto the level of the original call.

2. **Recursion Terminating Condition**: Every recursive function must have a terminating condition.
   In the above example the terminating condition is chosen as `n==1`. If the terminating condition is missing, then the recursive function would keep calling itself an infinite number of times.

Examples:

- Factorial:

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$$

```
function outfact = factrec(n)
% factrec: calculates the factorial of an integer using recursion
if n<1
    error('Error: n must be positive \n');
elseif n==1
    outfact = 1;
else
    outfact = factrec(n-1) * n;
end
```

- Calculation of a Power series:
  For example, the exponential of a real number `x` can be obtained using the following infinite sum:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots + \frac{x^n}{n!} + \cdots$$

  A *recursive* MATLAB user defined function that takes as input the values of `x` and `n` and returns two outputs `[factn expxn]` which are $n!$ and e^x upto n terms can be implemented as:

```
function [factn expxn]= recexp(x,n)
%This function takes inputs (x,n) to calculate n! and e^x with n terms and outputs

if n<0 || n ~=round(n)
    error('Error: n must be a non-negative integer');
elseif n==0    %terminating condition
    factn = 1;
    expxn = 1;
else
    %recursive call for (n-1)! and e^x with n-1 terms
    [fnm1 expnm1] = recexp(x,n-1);

    factn = fnm1*n;                  % get n!
    expxn = expnm1 + x^n/factn;      % get e^x with n terms
end
```

  Note that we combined multiple outputs in the above recursive function for better efficiency (instead of having different recursive calls for summing and factorial).

- Fibonacci numbers: 0 1 1 2 3 5 8 13 21 ...

$$F(n) = F(n-1) + F(n-2)$$

where $F(0) = 0$ and $F(1) = 1$.

```
function [outfn] = fiborec(n)
% fiborec: calculates the first n fibonacci numbers
if n<1
    error('Error: n must be positive \n');
elseif n==1
    outfn = 0;
elseif n==2
    outfn = [0 1];
else
    fnm1 = fiborec(n-1);
    outfn = fnm1(n-1) + fnm1(n-2);
    outfn = [fnm1 outfn];
end
```

The following MATLAB code computes and displays the first n Fibonacci numbers using Loops and Recursion:

```
clear all;
clc;
n = input('Enter the number of Fibonacci numbers required: ');

fprintf('Fibonacci numbers with Loops: \n');
for ii = 1:n
    if ii==1
        fn = 0;
    elseif ii==2
        fnm1 = 0;
        fn = 1;
    else
        fnew = fn + fnm1;
        fnm1 = fn;
        fn = fnew;
    end
    fprintf('%d ', fn);
end
fprintf('\n');

fprintf('Fibonacci numbers with Recursion: \n');
fprintf('%d ',fiborec(n))
fprintf('\n');
```

Chapter 7

External File Input/Output

Previously, we have learned some of the Matlab input/output functions, such as loading and saving Matlab data using the `load` and `save` commands, and writing out formatted data using the `fprintf` function. In this chapter, we will learn more about Matlab's input/output capabilities.

7.1 The `textread()` Function

The `textread` function reads ASCII files that are formatted into columns of data, where each column can be of a different type, and stores the contents of each column in a separate output array. The form of the `textread` function is

```
[a, b, c, ...]  = textread(filename, format, n)
```

where `filename` is the name of the file to open, `format` is a string containing a description of the type of data in each column, and `n` is the number of lines to read. If `n` is missing, the function reads to the end of the file. The format string contains the same types of format descriptors as function `fprintf`. Note that the number of output arguments must match the number of columns that you are reading.

For example, suppose that file `input.dat` contains the following data:

```
James Jones O+ 3.51 22 Yes
Sally Smith A+ 3.28 23 No
```

This data could be read into a series of arrays with the following function:

```
[first,last,blood,gpa,age,answer] = textread('input.dat','%s %s %s %f %d %s')
```

This function can also skip selected columns by adding an asterisk to the corresponding format descriptor (for example, `%*s`). The following statement reads only the first name, last name, and gpa from the file:

```
[first,last,gpa] = textread('input.dat','%s %s %*s %f %*d %*s')
```

Function `textread` is much more flexible than the `load` command. The `load` command assumes that all of the data in the input file is of a single type – it cannot support different types of data in different columns. In addition, it stores all of the data into a single array. In contrast, the `textread` function allows each column to go into a separate variable, which is much more convenient when working with columns of mixed data.

7.2 Introduction to MATLAB File Processing

To use external files within a MATLAB program, we need some way to select the desired file and to read from or write to it. MATLAB has a very flexible method to read and write files, whether they are on disk, magnetic tape, or some other device attached to the computer. This mechanism is called the **file identifier**, `(fid)`. The file identifier is a positive integer number that MATLAB assigns to a file when it is opened, and used for all reading, writing, and control operations on that file. File identifiers are assigned to disk files or devices using the `fopen()` statement, and are detached from them using the `fclose()` statement. Once an external file is opened using the `fopen()` statement and assigned a file id, we can read and write to that file using MATLAB file input and output statements. When we are done with the file, the `fclose()` statement closes the file and makes the file id invalid. The `frewind()` and `fseek()` statements may be used to change the current reading or writing position in a file while it is open.

Data can be written to and read from files in two possible ways: as **formatted text** (character) data or as **binary** data.

- Data in formatted files is translated into characters that can be read directly by a user. However, formatted I/O operations are slower and less efficient than binary I/O operations.
- Binary data consists of the actual bit (0,1) patterns that are used to store the data in computer memory. Reading and writing binary data is very efficient, but a user cannot read the data stored in the file.

We will discuss both types of I/O operations later in this chapter.

7.3 File Opening and Closing

7.3.1 The `fopen` Function

Then `fopen` function opens a file and returns a file id number that MATLAB has assigned to it. The basic forms of this statement are

```
fid = fopen(filename, permission)
[fid, message] = fopen(filename, permission)
[fid, message] = fopen(filename, permission, format)
```

where `filename` is a string specifying the name of the file to open, `permission` is a character string specifying the mode (read, write or append) in which the file is opened, and `format` is

an optional string specifying the numeric format of the data in the file. The `format` option is rarely used. If the open is successful, `fid` will contain a positive integer after this statement is executed, and `message` will be an empty string. If the open fails, `fid` will contain a `-1` after this statement is executed, and `message` will be a string explaining the error. The possible permission strings are shown in the following table.

`fopen()` File Permissions

File Permission	Meaning
`'r'`	Open an existing file for reading only (default).
`'r+'`	Open an existing file for reading and writing.
`'w'`	Delete the contents of an existing file (or create a new file) and open it for writing only.
`'w+'`	Delete the contents of an existing file (or create a new file) and open it for reading and writing.
`'a'`	Open an existing file (or create a new file) and open it for writing only, appending to the end of the file.
`'a+'`	Open an existing file (or create a new file) and open it for reading and writing, appending to the end of the file.

On some platforms such as PCs, it is important to distinguish between text files and binary files. If a file is to be opened in text mode, then a `t` should be added to the permissions string (for example, `'rt'` or `rt+`). If a file is to be opened in binary mode, a `b` may be added to the permissions string (for example, `'rb'`), but this is not actually required, since files are opened in binary mode by default. This distinction between text and binary files does not exist on Unix or Linux computers, so the `t` and `b` is never needed on those systems. Some examples of correct `fopen` functions are shown below.

Case 1: Opening a Binary File for Input

The function below opens a file named `example.dat` for binary input only.

```
fid = fopen('example.dat', 'r')
```

The permission string is `'r'`, indicating that the file is to be opened for reading only.

Case 2: Opening a File for Text Output

The function below open a file named `outdat` for text output only.

```
fid = fopen('outdat', 'wt')
```
or
```
fid = fopen('outdat', 'at')
```

The `'wt'` permissions string specifies that the file is a new text file; if it already exists, the old file will be deleted and a new empty file will be opened for writing. This is the

proper form of the `fopen` function for an *output file* is we want to replace pre-existing data. The `'at'` permissions string specifies that we want to append to an existing text file. If it is already exists, then it will be opened and new data will be appended to the currently existing information. This is the proper form of the `fopen` function for an *output file* if we don't want replace pre-existing data.

Case 3: Opening a Binary File for Read/Write Access

The function below opens a file named `binexample` for binary input and output.

```
fid = fopen('binexample', 'r+')
```

The function below also opens the file for binary input and output.

```
fid = fopen('binexample', 'w+')
```

The difference between the first and the second statements is that the first statement required the file to exist before it is opened, while the second statement will delete any pre-existing file.

Always be careful to specify the proper permissions in `fopen` statements, depending on whether you are reading from or writing to a file. This practice will help prevent errors such as accidentally overwriting data files that you want to keep. It is also important to check for errors after you attempt to open a file. If the `fid` is `-1`, then the file failed to open. You should report this problem to the user, and either select another file or else quit the program.

7.3.2 The `fclose` Function

The `fclose` function closes a file. Its form is

```
status = fclose(fid)
status = fclose('all')
```

where `fid` is a file id and `status` is the result of the operation. If the operation is successful, `status` will be `0`; if it is unsuccessful, `status` will be `-1`. The form `status = fclose('all')` closes all opened files. It returns a status of `0` if all files close successfully, and `-1` otherwise.

7.4 File Positioning and Status Functions

- **exist**:
  Check whether an item of a specified *kind* (variable, inbuilt, file) and specified *name* exists or not. It returns 2 if a 'file' of the given name exists. Its form is:

  `status = exist(`*`name, kind`*`)`

  Example: `stat = exist('fileio.txt','file')`

- **feof**:
  Returns logical true(1) or false(0) depending upon whether the end of file for file id `fid` has been reached or not.

  ```
  eofstat = feof(fid)
  ```

- **frewind**:
  To reset the file position at the beginning of the file.

  ```
  frewind(fid)
  ```

- **ftell**:
  Returns the current position within a file using a non-negative integer denoting bytes from the beginning.

  ```
  position = ftell(fid)
  ```

- **fseek**:
  To relocate the file position by `offset` number of bytes relative to a position specified by `origin` which can be `('bof', 'cof', 'eof')` denoting beginning of file, current position in file and end of file.

  `status = fseek(fid,` *`offset, origin`*`)`

- **ferror**:
  Returns the error status number `errnum` and `msg` of the most recent file I/O operation associated with the specified file id `fid`.

  ```
  [msg errnum] = ferror(fid)
  ```

7.5 I/O Functions for Formatted Text Data

7.5.1 The `fprintf` Function

The `fprintf` function writes formatted data in a user-specified format to a file. Its form is

```
count = fprintf(fid, format, val1, val2, ...)
fprintf(format, val1, val2, ...)
```

where `fid` is the file id of a file to which the data will be written, and `format` is the format string controlling the appearance of the data. If `fid` is missing, the data is written to the standard output device (the Command Window). This is the form of `fprintf` that we have learned before.

The format string specifies the alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters along with special sequences of characters that specify the exact format in which the output data will be displayed. A single % character always marks the beginning of a format. If an ordinary % is to be printed out, then it must appear in the format string as %%. After the % character, the format can have a flag, a field-width and precision specifier, and a conversion specifier. The % character and the conversion specifier are always required in any format, while the field, field-width, and precision specifier are optional.

The possible conversion specifiers are listed in the following table.

Format Conversion Specifiers for `fprintf`

Specifier	Description
`'%c'`	Single character
`'%s'`	String of characters
`'%d'`	Decimal notation (signed)
`'%u'`	Decimal notation (unsigned)
`'%e'`	Exponential notation using a lowercase `e`
`'%E'`	Exponential notation using an uppercase `E`
`'%f'`	Fixed-point notation
`'%g'`	The more compact of `%e` or `%f`
`'%G'`	Same as `%g`, but using an uppercase `E`
`'%o'`	Octal notation (unsigned)
`'%x'`	Hexadecimal notation (using lowercase letters `a-f`)
`'%X'`	Hexadecimal notation (using uppercase letters `A-F`)

If a field width and precision are specified in a format, then the number before the decimal point is the field width, which is the number of characters used to display the number. The number after the decimal point is the precision, which is the minimum number of significant digits to display after the decimal point.

Make sure that there is a one-to-one correspondence between the types of the data in an `fprintf` function and the types of the format conversion specifiers in the associated format string; otherwise, your program will produce unexpected results.

7.5.2 The `fscanf` Function

The `fscanf` function reads formatted data in a user-specified format from a file:

```
array = fscanf(fid, format)
[array, count] = fscanf(fid, format, size)
```

where `fid` is the file id of a file from which the data will be read, `format` is the format string controlling how the data is read, and `array` is the array that receives the data. The output argument `count` returns the number of values read from the file.

The optional argument `size` specifies the amount of data to be read from the file. There are three versions of this argument:

- `n` – Read exactly `n` values. After this statement, `array` will be a column vector containing `n` values read from the file.
- `Inf` – Read until the end of the file. After this statement, `array` will be a column vector containing all of the data until the end of the file.
- `[m, n]` – Read exactly `m` $\times$ `n` values, and format the data as an `m` $\times$ `n` array.

The format string specifies the format of the data to be read. It can contain ordinary characters along with format conversion specifiers. The `fscanf` function compares the data in the file with the format conversion specifiers in the format string. As long as the two match, `fscanf` converts the value and stores it in the output array. The process continues until the end of the file or until the amount of data in `size` has been read, whichever comes first. If the data in the file does not match the format conversion specifiers, the operation of `fscanf` stops immediately. The format conversion specifiers for `fscanf` are basically the same as those for `fprintf`.

7.5.3 The `fgetl` and `fgets` Functions

The `fgetl` and `fgets` functions read the next line of input from the file specified by file id `fid` and return a character array (string), `line` as output. Their general form is

```
line = fgetl(fid)
line = fgets(fid)
```

The difference is that for `fgetl()`, the output string `line` *does not* contain the end of line character in the end where as for `fgets()`, it *does*. If an end of a file is encountered, the value of `line` is set to `-1`.

7.6 I/O Functions for Binary Data

7.6.1 The `fwrite` Function

The `fwrite` function writes binary data in a user-specified format to a file. Its form is

```
count = fwrite(fid, array, precision)
count = fwrite(fid, array, precision, skip)
```

where `fid` is the file id of a file opened with the `fopen` function, `array` is the array of values to write out, and `count` is the number of values written to the file.

Matlab writes out data in `column order`, which means that the entire first column is written out, followed by the entire second column, etc. For example, if

$$\texttt{array} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix},$$

the data will be written out in the order `1, 3, 5, 2, 4, 6`.

The optional `precision` string specifies the format in which the data will be output. Matlab supports both platform-independent precision strings, which are the same for all computers that Matlab runs on, and platform-dependent precision strings, which vary among different types of computers. The possible platform-independent precisions are presented in the following table. All of these precisions work in units of bytes, except for `'bitN'` or `'ubitN'`, which work in units of bits.

Selected Matlab Precision Strings

Matlab Precision String	C/Fortran Equivalent	Meaning
`'char'`	`'char*1'`	8-bit character
`'schar'`	`'signed char'`	8-bit signed character
`'uchar'`	`'unsigned char'`	8-bit unsigned character
`'int8'`	`'integer*1'`	8-bit integer
`'int16'`	`'integer*2'`	16-bit integer
`'int32'`	`'integer*4'`	32-bit integer
`'int64'`	`'integer*8'`	64-bit integer
`'uint8'`	`'integer*1'`	8-bit unsigned integer
`'uint16'`	`'integer*2'`	16-bit unsigned integer
`'uint32'`	`'integer*4'`	32-bit unsigned integer
`'uint64'`	`'integer*8'`	64-bit unsigned integer
`'float32'`	`'real*4'`	32-bit floating point
`'float64'`	`'real*8'`	64-bit floating point
`'bitN'`		N-bit signed integer, 1 <= N <= 64
`'ubitN'`		N-bit unsigned integer, 1 <= N <= 64

The optional argument `skip` specifies the number of bytes to skip in the output file before each write. Note that if `precision` is a bit format like `'bitN'` or `'ubitN'`, skip is specified in bits instead of bytes.

7.6.2 The `fread` Function

The `fread` function reads binary data in a user-specified format from a file, and returns the data in a (possibly different) user-specified format. Its form is

```
[array, count] = fread(fid, size, precision)
[array, count] = fread(fid, size, precision, skip)
```

where `fid` is the file id of a file opened with the `fopen` function, `size` is the number of values to read, `array` is the array to contain the data, and `count` is the number of values read from the file.

The optional argument `size` specifies the amount of data to be read from the file. There are three versions of this argument:

- `n` – Read exactly `n` values. After this statement, `array` will be a column vector containing `n` values read from the file.
- `Inf` – Read until the end of the file. After this statement, `array` will be a column vector containing all of the data until the end of the file.
- `[m, n]` – Read exactly `m` $\times$ `n` values, and format the data as an `m` $\times$ `n` array.

The `precision` argument specifies both the format of the data on the disk and the format of the data array to be returned to the calling program. The general form of the precision string is

```
'disk_precision => array_precision'
```

where `disk_precision` and `array_precision` are both among the precision strings found in the previous table. The `array_precision` value can be defaulted. If it is missing, then the data is returned in a `double` array. There is also a shortcut form of this expression if the disk precision and the array precision are the same: `'*disk_precision'`.

Chapter 8

Numerical Methods in MATLAB

8.1 Matrix Algebra

One of the most common linear algebra problems is the solution of a linear set of equations. For example, consider the set of equations

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix}$$

or

$$\mathbf{Ax} = \mathbf{y}$$

The above equations define the product of the $n \times n$ matrix $\mathbf{A}$ and the $n \times 1$ vector $\mathbf{x}$ as being equal to the $n \times 1$ vector $\mathbf{y}$. The existence of solution to the above equation is a fundamental problem in linear algebra. Moreover, when a solution does exist, there are numerous approaches to finding the solution, such as Gaussian elimination, LU factorization, or direct use of $\mathbf{A}^{-1}$. It is beyond the scope of this text to discuss the many analytical and numerical issues of matrix algebra. We will only demonstrate how Matlab can be used to solve problems like the one above.

$\mathbf{Ax} = \mathbf{y}$ has a unique solution if the rank of $\mathbf{A}$ and the rank of the augmented matrix $[\mathbf{A} \quad \mathbf{y}]$ are both equal to n. Alternatively, one can check the condition number of $\mathbf{A}$. If the condition number is not excessively large, then $\mathbf{A}$ has an inverse with good numerical properties, which means we can find $\mathbf{A}^{-1}$. The rank of a matrix can be calculated by using Matlab function `rank`. And the condition number of a matrix can be obtained by using Matlab function `cond`. For example, `rank(A)`, `rank[A y]`, and `cond(A)` give the rank of $\mathbf{A}$, the rank of $[\mathbf{A} \quad \mathbf{y}]$, and the condition number of $\mathbf{A}$, respectively.

There are two ways to find the solution of $\mathbf{Ax} = \mathbf{y}$ in Matlab:

1. `x = inv(A)*y`. This method is more straightforward while less favorable. Please note `inv(A)` computes $\mathbf{A}^{-1}$.

2. `x = A\y`. The method using matrix left division is preferable. This method utilizes an LU factorization approach, which requires fewer floating-point operations and as a result is significantly faster.

In addition to the solution of linear sets of equations, Matlab offers numerous matrix functions that are useful for solving numerical linear algebra problems. Some of these functions are listed below:

- `chol(A)` – Cholesky factorization.
- `cond(A)` – Matrix condition number.
- `det(A)` – Determinant.
- `eig(A)` – Vector of eigenvalues.
- `[V, D] = eig(A)` – Matrix of eigenvectors, and diagonal matrix containing eigenvalues.
- `inv(A)` – Matrix inverse.
- `[L, U] = lu(A)` – LU decomposition.
- `norm(A, type)` – Matrix and vector norms.
- `rank(A)` – Matrix rank.
- `schur(A)` – Schur decomposition.
- `svd(A)` – Singular values.
- `[U, S, V] = svd(A)` – Singular value decomposition.
- `trace(A)` – Sum of matrix diagonal elements.

8.2 Data Analysis

Matlab provides numerous functions that performs statistical analyses on data sets. Some of these functions are listed below:

- `max(x)` – If `x` is a vector then the function returns the largest value in `x`. If `x` is a matrix then the function returns a row vector containing the maximum element from each column.
- `[a, b] = max(x)` – In this form, if `x` is a vector, then the function stores the maximum value of `x` in a scalar `a`, and the index of the maximum value in the scalar `b`. If there are several identical maximum values, the index of the first one found is returned. If `x` is a matrix, then the function stores the maximum values of `x` in a vector `a` and the indices of the maximum values in a vector `b`.
- `max(x,[ ],1)` – The function returns a row vector containing the maximum element from each column.

- `max(x,[ ],2)` – The function returns a column vector containing the maximum element from each row.
- Replacing `max` with `min` in the above functions gives the corresponding functions for finding minimums.
- `mean(x)` – If `x` is a vector, this function computes the average value of the elements of the vector `x`. If `x` is a matrix, this function computes a row vector that contains the average value of each column.
- `mean(x,1)` – The function returns a row vector containing the average value from each column.
- `mean(x,2)` – The function returns a column vector containing the average value from each row.
- `median(x)` – If `x` is a vector, this function computes the median value of the elements of the vector `x`. If `x` is a matrix, this function computes a row vector that contains the median value of each column.
- `median(x,1)` – The function returns a row vector containing the median value from each column.
- `median(x,2)` – The function returns a column vector containing the median value from each row.
- `sum(x)` – If `x` is a vector, this function returns the sum of the elements of `x`. If `x` is a matrix, this function returns a row vector that contains the sum of each column.
- `sum(x,1)` – The function returns a row vector containing the sum of elements from each column.
- `sum(x,2)` – The function returns a column vector containing the sum of elements from each row.
- `sort(x)` – If `x` is a vector then this function sorts the values into ascending order. If `x` is a matrix then this function sorts each column into ascending order.
- `[a, b] = sort(x)` – The sorted values are returned to the matrix `a` and reordered indices are stored in the matrix `b`.
- `std(x)` – If `x` is a vector, this function computes the standard deviation of the values of `x`. If `x` is a matrix, this function computes a row vector containing the standard deviation of each column.
- `var(x)` – If `x` is a vector, this function computes the variance of the values of `x`. If `x` is a matrix, this function computes a row vector containing the variance of each column.

8.3 Polynomials

Matlab provides a number of functions for manipulating polynomials. We will introduce a couple of those functions.

8.3.1 Roots

Finding the roots of a polynomial is a problem common to many disciplines. Matlab solves this problem and finds the roots of a polynomial by using function `roots`. In Matlab a polynomial is represented by a row vector of its coefficients in descending order. For example, the polynomial $x^4 - 12x^3 + 0x^2 + 25x + 116$ is represented by a vector

```
p = [1 -12 0 25 116];
```

Note that terms with zero coefficients must be included. Matlab has no way of knowing which terms are zero unless you specifically identify them. Given this form, the roots can be found by

```
r = roots(p);
```

where `r` is a column vector containing the roots of the polynomial. **Matlab adopts the convention that polynomials are row vectors and roots are column vectors**.

8.3.2 Curve Fitting

In numerous application areas, one is faced with the task of fitting a curve to measured data. Sometimes the chosen curve passes through the data points, but at other times the curve comes close to, but does not necessarily pass through the data points. In the most common situation, the curve is chosen so that the sum of the squared errors at the data points is minimized. This choice results in a **least squares** curve fit. It is straightforward and common to use a polynomial as the basis function for the least squares curve fitting.

In Matlab, the function `polyfit` solves the least squares polynomial curve-fitting problem. To illustrate the use of this function, let's start with the data

```
x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0];
y = [-0.45 1.98 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.20];
```

To use `polyfit`, we must give it the above data and the order or degree of the polynomial we wish to best fit the data. If we choose $n = 1$ as the order, the best straight-line approximation will be found. This is often called **linear regression**. If we choose $n = 2$ as the order, a quadratic polynomial will be found. For example, let's choose a quadratic polynomial:

```
n = 2;
p = polyfit(x, y, n);
```

The output of `polyfit` is a row vector of the polynomial coefficients, which are `-9.8147 20.1338 -0.0327` stored in `p`. This means the solution is $y(x) = -9.8147x^2 + 20.1338x - 0.0327$. Figure 8.1, which is created by the following code, shows the original data and the fitted curve.

```
xi = linspace(0, 1, 100);
yi = polyval(p, xi);
figure(1)
plot(x, y, '-o', xi, yi, '--')
xlabel('x'); ylabel('y = f(x)');
title('Second Order Curve Fitting');
```

where function `polyval` is used to evaluate the polynomial at a set of data points.

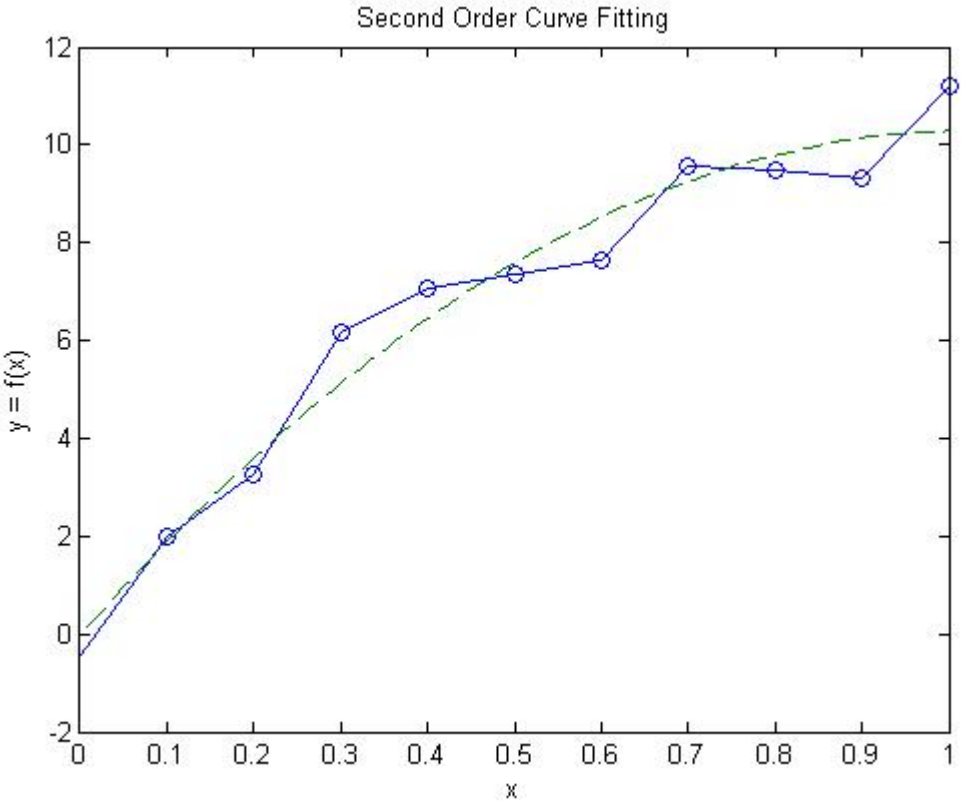

Figure 8.1: Original data points and the second order curve fitting.

8.4 Integration

Matlab provides functions for numerically approximating the integral of a function. These functions are `quad`, `quadl`, and `dblquad`.

- `quad` – Numerically evaluate integral, adaptive Simpson quadrature.
- `quadl` – Numerically evaluate integral, adaptive Lobatto quadrature.
- `dblquad` – Numerically evaluate double integral.

$f(x) = e^{-x^2}$ occurs frequently in physics problems but cannot be integrated analytically. Let's show how to use function `quad` to compute $\int_0^1 f(x)dx$ numerically.

First, we need to define the function to be integrated. The function must support a vector input argument and return a vector of outputs for a vector inputs. For example, $f(x) = e^{-x^2}$ is defined in `myfun.m` as

```
function y = myfun(x)
y = exp(-x.^2);
```

Then, we may use `quad` to compute the integration.

```
z = quad(@myfun, 0, 1);
```
or
```
z = quad('myfun', 0, 1);
```

where the first argument is the defined function to be integrated, the second and third arguments are the lower limit and upper limit of integration, respectively.

8.5 Differential Equations

Matlab has the capability to solve as wide variety of problems involving differential equations, such as ordinary differential equations (ODEs), initial value problems (IVPs), boundary value problems (BVPs), and partial differential equations (PDEs).

8.5.1 IVP Format

Matlab computes the time history of a set of coupled first-order differential equations with known initial conditions. These problems are called initial value problems and have the form

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

which is vector notation for the set of differential equations

$$\begin{array}{ll} \dot{y}_1 = f_1(t, y_1, y_2, \ldots, y_n) & y_1(t_0) = y_{10} \\ \dot{y}_2 = f_2(t, y_1, y_2, \ldots, y_n) & y_2(t_0) = y_{20} \\ \vdots & \vdots \\ \dot{y}_n = f_n(t, y_1, y_2, \ldots, y_n) & y_n(t_0) = y_{n0} \end{array}$$

where $\dot{y}_i = dy_i/dt$, n is the number of first-order differential equations, and y_{i0} is the initial condition associated with the ith equation. When an initial value problem is not specified as a set of first-order differential equations, it must be rewritten as one. For example, consider the classic van der Pol equation

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$$

where μ is a parameter greater than zero. If we choose $y_1 = x$ and $y_2 = dx/dt$, the van der Pol equation becomes

$$\dot{y}_1 = y_2$$
$$\dot{y}_2 = \mu(1 - y_1^2)y_2 - y_1$$

We will demonstrate Matlab ODE solvers by solving this problem.

8.5.2 ODE Solvers

Matlab offers seven initial value problem solvers. We will talk about two of them.

- `ode45` – An explicit one-step Runge Kutta medium-order (4th- to 5th-order) solver. Suitable for non-stiff problems that require moderate accuracy. **This is typically the first solver to try on a new problem**.
- `ode15s` – An implicit, multistep numerical differentiation solver of varying order (1st- to 5th-order). Suitable for stiff problems that require moderate accuracy. **This is typically the solver to try if `ode45` fails or is too inefficient**.

8.5.3 Basic Use

Before a set of differential equations can be solved, they must be coded in a function M-file. That is, the file must accept a time `t` and a solution `y` and return values for the derivatives. For the van del Pol equation, this ODE file can be written as follows.

```
function [ydot] = vdpol(t, y)
% This function (ydot=f(t,y)) defines the system of ODEs you are trying to solve.
% The input 'y' can be a Nx1 vector; Thus 'ydot' must also be Nx1 vector
mu = 2; % some constant
y1dot = y(2);
y2dot = mu*(1-y(1)^2)*y(2) - y(1);
ydot = [ y1dot; y2dot];
```

Note that the input arguments are `t` and `y` but that the function does not use `t`. Note also that the output `ydot` must be a column vector.

Given the above ODE file, this set of ODEs is solved by the following statements.

```
tspan = [0 20]; % time span to integrate over
y0 = [2; 0]; % initial conditions (must be a column)
[t, y] = ode45('vdpol', tspan, y0);
plot(t,y(:,1),'b.-',t,y(:,2),'g.-')
legend('y','dy/dt')
```

where `t` contains the time points, `y(:,1)` contains the solution of `x(t)`, and `y(:,2)` contains the values of `dx/dt`.

Example 1: In a dynamic problem, we have the following governing equation for a bowl sliding down a rotating support as shown in 8.2.

$$\ddot{r} - \omega_0^2 r = -g\ sin(\theta_0 + \omega_0 t)$$

where r is the distance of bowl away from the support end. Initial distance $L = 2\text{m}$; Angular rotating speed of slope $\omega_0 = 0.1$ rad/s ; Initial angle of the slope $\theta_0 = 0.2$ rad. g is the acceleration due to gravity. Solve for $r(t)$ and $\dot{r}(t)$ using Matlab. Verify with the exact solution given by:

$$\begin{aligned} r(t) &= \left(L - \frac{g\sin\theta_0}{2\omega_0^2}\right)\cosh(\omega_0 t) - \frac{g\cos\theta_0}{2\omega_0^2}\sinh(\omega_0 t) + \frac{g}{2\omega_0^2}\sin(\theta_0 + \omega_0 t) \\ \dot{r}(t) &= \left(L\omega_0 - \frac{g\sin\theta_0}{2\omega_0}\right)\sinh(\omega_0 t) - \frac{g\cos\theta_0}{2\omega_0}\cosh(\omega_0 t) + \frac{g}{2\omega_0}\cos(\theta_0 + \omega_0 t) \end{aligned}$$

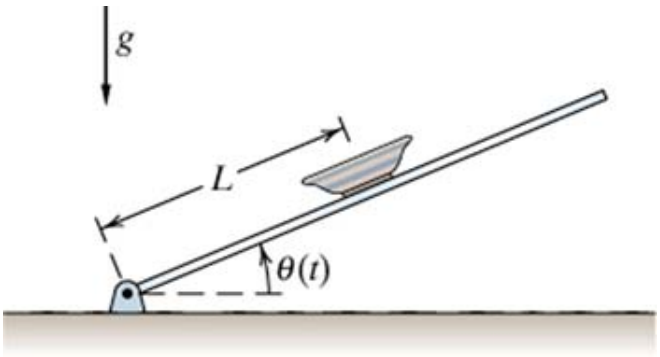

Figure 8.2: Object on rotating slope.

Solution
First we need convert this second order ODE to a set of first-order differential equations. Let $y_1 = r$ and $y_2 = dr/dt$, we have

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= \omega^2 y_1 - g\ sin(\theta_0 + \omega_0 t) \end{aligned}$$

The code for the derivative function as well as the statement for solving the ODEs are listed below.

```
function ydot = bowlonslope(t, y)

w0 = 0.1; theta0 = 0.2; g = 9.8;
ydot = [y(2); w0^2*y(1) - g*sin(theta0+w0*t)];
```

AND

```
clear all; close all;

L = 2.0; tspan = [0 1.0]; y0 = [L; 0]; g = 9.8; w0 = 0.1;
theta0 = 0.2;

[t, y] = ode45('bowlonslope', tspan, y0);

ti = linspace(0, 1.0, 100);
ri = (L - g/2/w0^2*sin(theta0))*cosh(w0*ti) - ...
g/2/w0^2*cos(theta0)*sinh(w0*ti) + g/2/w0^2*sin(theta0+w0*ti);
ridot = (L*w0 - g/2/w0*sin(theta0))*sinh(w0*ti) - ...
g/2/w0*cos(theta0)*cosh(w0*ti) + g/2/w0*cos(theta0+w0*ti);

figure(1)
plot(t, y(:,1), 'b.-', ti, ri, 'k-');
xlabel('t');
ylabel('distance r');
legend('Matlab Numerical Solution','Analytical Exact Solution')

figure(2)
plot(t, y(:,2), 'b.-', ti, ridot, 'k-');
xlabel('t');
ylabel('bowl speed');
legend('Matlab Numerical Solution','Analytical Exact Solution')
```

Figure 8.3 shows the distance r versus time t. In this figure, analytical solution is also given for comparison. Figure 8.4 shows the speed of the bowl $\dot{r}$ versus time t.

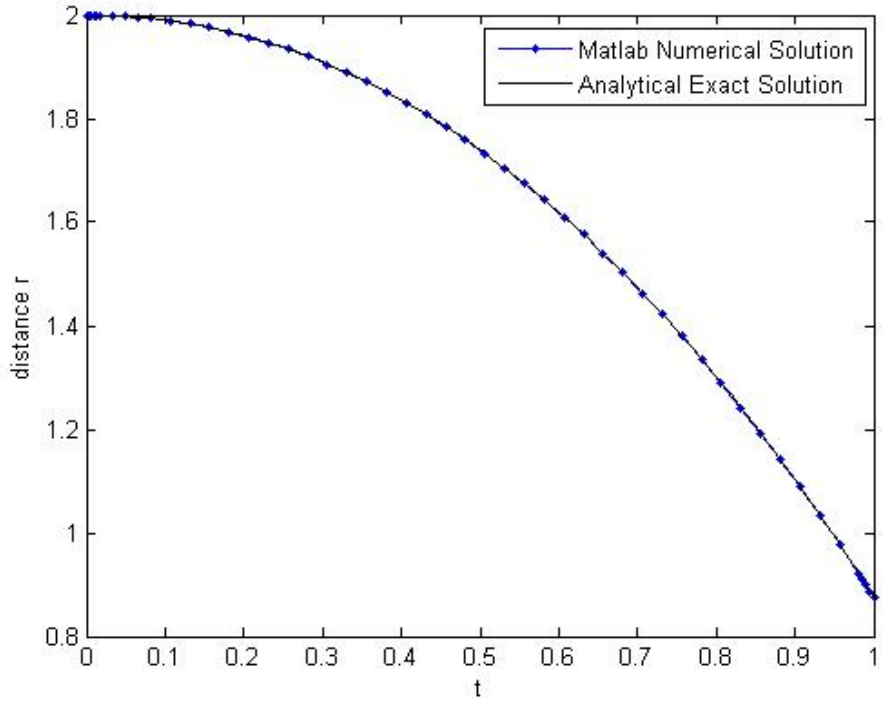

Figure 8.3: Distance versus time.

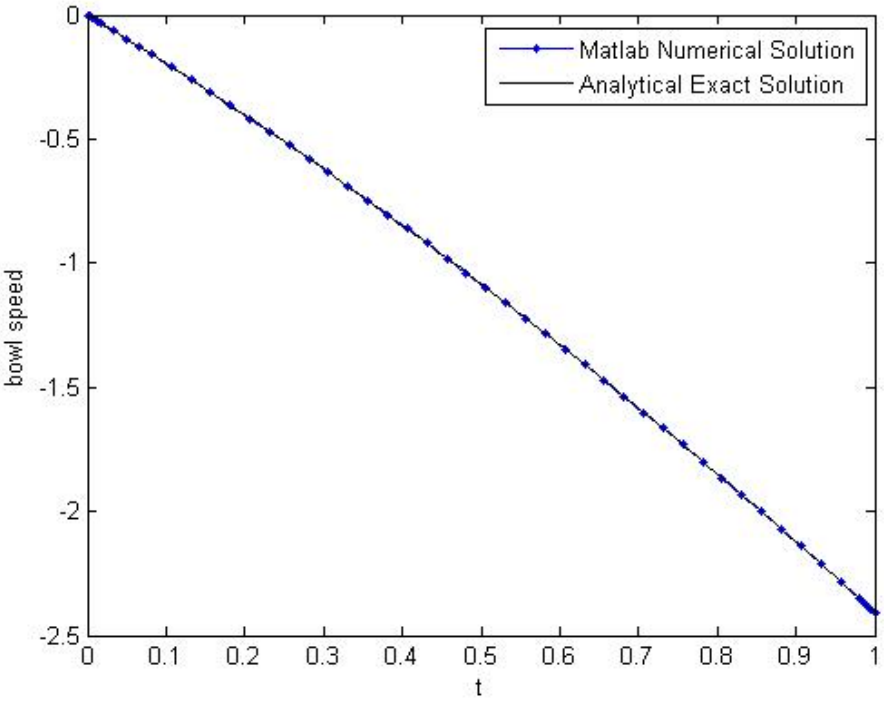

Figure 8.4: Speed versus time.

Example 2: In Chemical Engineering, the following set of differential equations describes the change in concentration three species in a tank. The reactions $A \rightarrow B \rightarrow C$ occur within the tank. The constants k_1, and k_2 describe the reaction rate for $A \rightarrow B$ and $B \rightarrow C$ respectively. The following ODEs are obtained:

$$\begin{aligned} \frac{dC_a}{dt} &= -k_1 C_a \\ \frac{dC_b}{dt} &= k_1 C_a - k_2 C_b \\ \frac{dC_c}{dt} &= k_2 C_b \end{aligned}$$

Where $k_1 = 1\ hr^{-1}$ and $k_2 = 2\ hr^{-1}$ and at time $t = 0$, $C_a = 5\ mol$ and $C_b = C_c = 0\ mol$. Solve the system of equations and plot the change in concentration of each species over time. Select a time interval [0 5] for the integration.

Solution:

The derivative function

```
function ydot = concentration(t, y)

k1 = 1; k2 = 2;
ydot = [-k1*y(1); k1*y(1)-k2*y(2); k2*y(2)];
```

The main program

```
clear all; close all;
tspan = [0 5]; y0 = [5 0 0];

[t, y] = ode45('concentration', tspan, y0);

figure(1)
plot(t, y(:,1), 'r-', t, y(:,2), 'b--', t, y(:,3), 'k*')
xlabel('Time (hr)');
ylabel('concentration of each species (mol/hr)');
legend('C_a','C_b','C_c','Location','East');
```

Figure 8.5 shows the result.

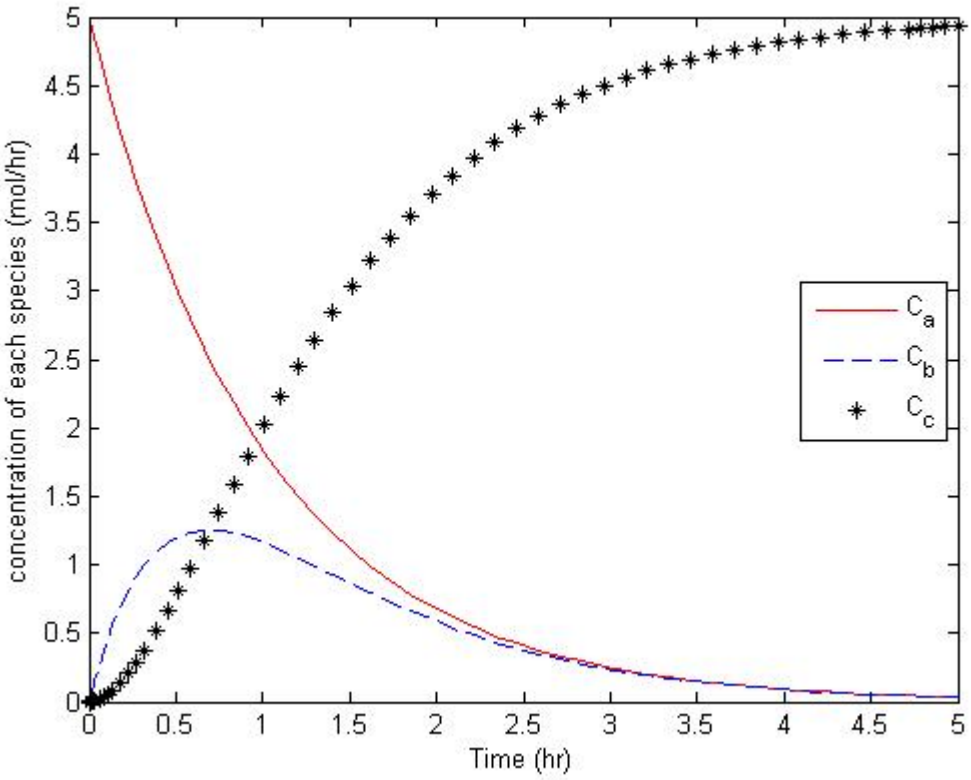

Figure 8.5: Concentrations versus time.

8.6 Advanced MATLAB Features

- Cell Arrays
- User defined structures and Classes
- Visual Programming with GUI
- Symbolic Math Toolbox
- Objects and Classes
- Parallel Programming

Chapter 9

Application to Civil Engineering: Structural Dynamics

In this chapter, a structural dynamics problem is presented and solved with Matlab. Suppose the dynamic behavior of a three-floor building, which is shown in Figure 9.1, needs to be determined. The building's displacement is measured at each floor. The columns separating the floors are modelled as springs and all the mass of the floors can be lumped into one value per floor. Additionally, velocity proportional dampers exert a small force on the lumped mass at a floor directly proportional to the velocity of that degree of freedom.

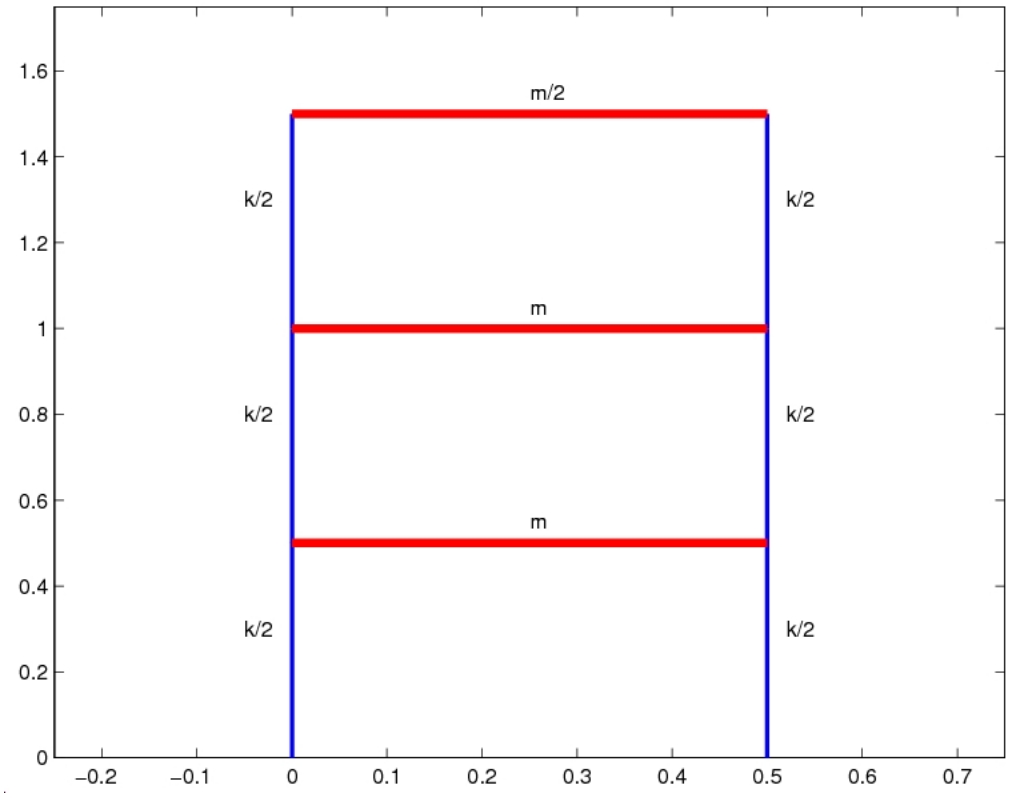

Figure 9.1: Three-floor building modelled as mass spring system.

The governing differential equation along with initial conditions for this system is

$$\boldsymbol{M}\,\ddot{\boldsymbol{x}}(t) + \boldsymbol{C}\,\dot{\boldsymbol{x}}(t) + \boldsymbol{K}\,\boldsymbol{x}(t) = \boldsymbol{F}(t)$$
$$\dot{\boldsymbol{x}}(0) = \boldsymbol{v}_0 \qquad \text{and} \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0$$

where $\boldsymbol{M}$ is the 3×3 mass matrix, $\boldsymbol{K}$ is the 3×3 stiffness matrix, and $\boldsymbol{C}$ is the 3×3 damping matrix. Vectors $\boldsymbol{x}$ and $\boldsymbol{F}$ store the floor displacements and applied loads at each floor, respectively. $\boldsymbol{x}_0$ and $\boldsymbol{v}_0$ give the initial state (displacement and velocity) of the structure at time $t = 0$.

Based on the given information about the mass of each floor and the stiffness of each column, The mass matrix, stiffness matrix and damping matrix can be constructed as

$$\boldsymbol{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m/2 \end{bmatrix};\ \boldsymbol{K} = \begin{bmatrix} 2k & -k & 0 \\ -k & 2k & -k \\ 0 & -k & k \end{bmatrix};\ C = \alpha\boldsymbol{M} + \beta\boldsymbol{K}$$

The above data can be defined in an input file **strdyninput.m**:

```
function [M C K L d u0 v0 Tspan deltaT] = strdyninput

% Initialization
m = 1.0; k = 1.0; L = 0.5; d = 0.5;
M = [m 0 0; 0 m 0; 0 0 m/2.0];
K = [2*k -k 0; -k 2*k -k; 0 -k k];
alpha = 0.0; beta = 0.0;
C = alpha*M + beta*K;

% Initial conditions
u0 = [0.05; 0.05; 0.05]; v0 = [0; 0; 0];
%u0 = [0; 0; 0]; v0 = [0.05; 0.05; 0.05];
%u0 = [0; 0; 0]; v0 = [0; 0; 0];

% Define time span and time step
Tspan = 25.0; deltaT = 0.1;
```

The externally applied force can also be defined as a function of time t in **strdynforce.m**:

```
function force = strdynforce(t)

force = [0; 0; 0];
%force = [-0.05*sin(t); 0.1*sin(t); 0.1*cos(t)];
```

Let's consider a free vibration problem, which has $\boldsymbol{F} = 0$. Initially, a rope pulled the first floor with a displacement s_0, as shown in Figure 9.2, and suddenly got cut at time $t = 0$. The initial conditions of this problem can be described as $\dot{\boldsymbol{x}}(0) = [0\ 0\ 0]^T$ and $\boldsymbol{x}(0) = [s_0\ s_0\ s_0]^T$.

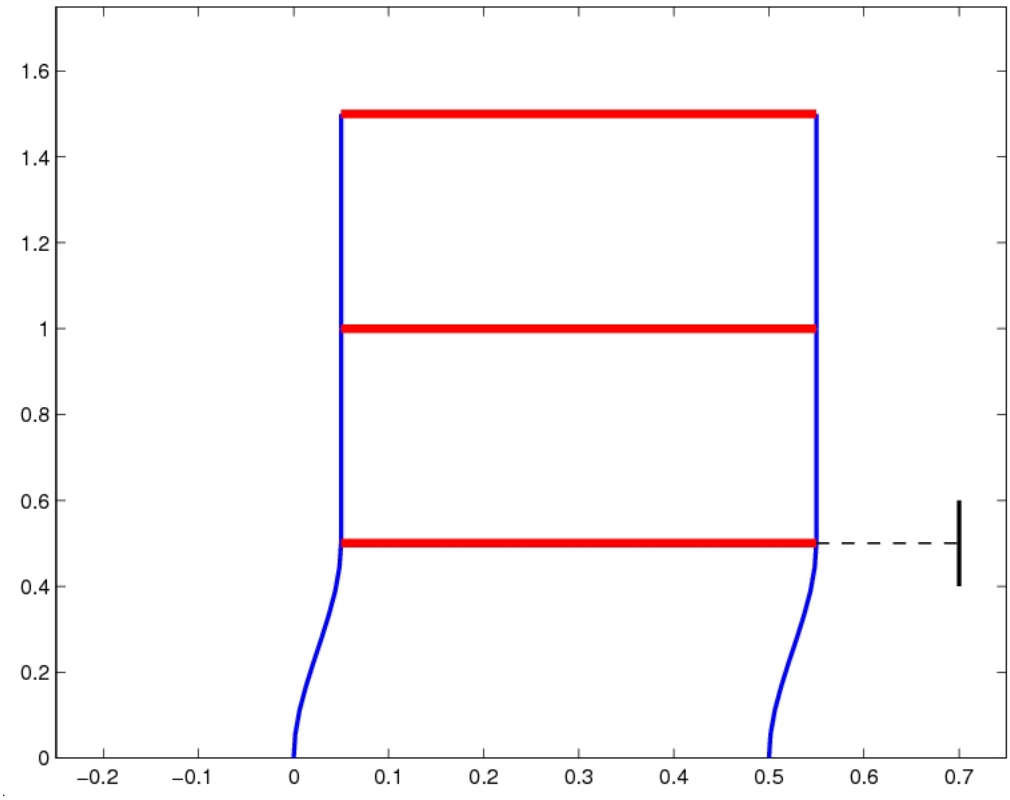

Figure 9.2: Initial configuration of the three-floor building.

The following two methods can be used to solve the Structural Dynamics problem:

Method 1: Using ode45()
Convert the given system into standard form:

$$\begin{aligned} \dot{\boldsymbol{y}} &= \boldsymbol{f}(t, \boldsymbol{y}) \\ \dot{\boldsymbol{y}}(t) &= \boldsymbol{f}(t, \boldsymbol{y}(t)) \end{aligned}$$

where $\boldsymbol{y}(t) = \begin{bmatrix} \boldsymbol{y}_1(t) \\ \boldsymbol{y}_2(t) \end{bmatrix}$, $\boldsymbol{y}_1(t) = \boldsymbol{x}(t)$; $\boldsymbol{y}_2(t) = \dot{\boldsymbol{y}}_1(t) = \dot{\boldsymbol{x}}(t)$ and

$$\begin{aligned} \dot{\boldsymbol{y}}_1(t) &= \boldsymbol{y}_2(t) \\ \dot{\boldsymbol{y}}_2(t) &= \boldsymbol{M}^{-1}\left[\boldsymbol{F}(t) - \boldsymbol{C}\,\boldsymbol{y}_2(t) - \boldsymbol{K}\,\boldsymbol{y}_1(t)\right] \end{aligned}$$

Now we can code this using `ode45()`. Define the standard form of the equation to be solved:
strdynfun.m:

```
function ydot = strdynfun(t, y)

[M C K] = strdyninput;
F = strdynforce(t);
```

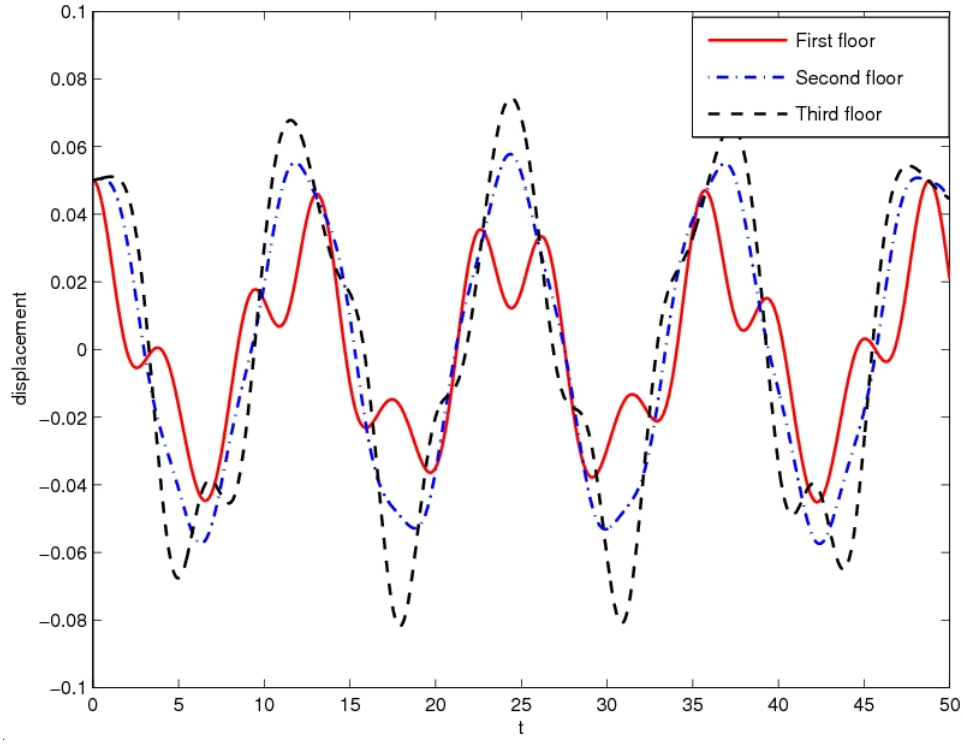

Figure 9.3: Displacements versus time.

```
n = size(M,1);
y1 = y(1:n);
y2 = y(n+1:2*n);

y1dot = y2;
y2dot = inv(M)*(F-K*y1-C*y2);
ydot = [y1dot; y2dot];
```

Main program **strdyn.m**

```
clear all; close all;
% Initialization
[M C K L d u0 v0 Tspan] = strdyninput;

[tmat, ymat] = ode45('strdynfun', [0 Tspan], [u0; v0]);

plot(tmat,ymat(:,1),'r', tmat,ymat(:,2),'b', tmat,ymat(:,3),'k');
xlabel('time');
ylabel('Displacement');
legend('Story 1','Story 2','Story 3','Location','SouthEast')
```

Method 2: Central difference method
The central difference method with a time step Δt is used for solving the second order differential governing equation. The calculation steps are listed below:

1. Calculate the vector value of initial acceleration, $\ddot{\boldsymbol{x}}_0$.

$$\ddot{\boldsymbol{x}}_0 = \boldsymbol{M}^{-1} * (\boldsymbol{F}_0 - \boldsymbol{C}\dot{\boldsymbol{x}}_0 - \boldsymbol{K}\boldsymbol{x}_0)$$

2. Compute one backward step (to time -1) by using the following equation.

$$\boldsymbol{x}_{-1} = \boldsymbol{x}_0 - \Delta t\dot{\boldsymbol{x}}_0 + \frac{(\Delta t)^2}{2}\ddot{\boldsymbol{x}}_0$$

3. Compute the effective stiffness matrix by using the following equation.

$$\boldsymbol{K}_{\text{eff}} = \left[\frac{1}{(\Delta t)^2}\boldsymbol{M} + \frac{1}{2\Delta t}\boldsymbol{C}\right]$$

4. **Loop** over time steps $i = 1{:}N$

   (a) Compute the next state (at t_{i+1}) of the system in terms of the two previous states t_i and t_{i-1}:

$$\boldsymbol{x}_{i+1} = \boldsymbol{K}_{\text{eff}}^{-1}\left[\boldsymbol{F}_i - \left(\boldsymbol{K} - \frac{2}{(\Delta t)^2}\boldsymbol{M}\right)\boldsymbol{x}_i - \left(\frac{1}{(\Delta t)^2}\boldsymbol{M} - \frac{1}{2\Delta t}\boldsymbol{C}\right)\boldsymbol{x}_{i-1}\right]$$

   (b) Store the solution and move to the next time step.

Main Program **strdyncd.m**

```
clear all; close all;

% Initialization
[M C K L d u0 v0 Tspan deltaT] = strdyninput;

% Define the time steps
t = -deltaT:deltaT:Tspan;
nsteps = length(t);

% Calculate the vector value of initial acceleration
F = strdynforce(0);
a0 = inv(M)*(F - C*v0 - K*u0);

% Define matrix storing the solutions
% The rows of the matrix are the degrees of freedom
```

```
% The columns of the array are the state of the system in time
Usol = zeros(3, nsteps);

% The second column is the position at t = 0
Usol(:,2) = u0(:);

% Compute one backward step (t = -\Delta t), and store x(-\Delta t) in the
% first column of the solution matrix
Usol(:,1) = u0(:) - deltaT*v0(:) + deltaT^2/2.0*a0(:);

% Compute the effective stiffness matrix and store the inverse
Keff = M/(deltaT^2) + C/(2.0*deltaT);
Keffinv = inv(Keff);

% March the equation forward, creating new steps by assigning the next
% state of the system as a function of the two previous states
for ii=3:nsteps
    F = strdynforce(t(ii));
    Usol(:,ii) = Keffinv*(F - (K-2.0*M/(deltaT^2))*Usol(:,ii-1) ...
                  - (M/(deltaT^2)-C/(2.0*deltaT))*Usol(:,ii-2));
end

% Plot the time history of the displacements of each floor
figure(1);
plot(t(2:nsteps), Usol(1,2:nsteps), 'r-', 'LineWidth', 1.5)
hold on
plot(t(2:nsteps), Usol(2,2:nsteps), 'b-', 'LineWidth', 1.5)
plot(t(2:nsteps), Usol(3,2:nsteps), 'k-', 'LineWidth', 1.5)
xlabel('time'); ylabel('Displacement');
legend('Story 1','Story 2','Story 3','Location','SouthEast')
pause;

% Animation
xp1 = linspace(0, L, 10);
xp2 = linspace(L, 2*L, 10);
xp3 = linspace(2*L, 3*L, 10);

coefMat1 = [0 0 0 1; (L)^3 (L)^2 L 1; 0 0 1 0; 3*(L)^2 2*(L) 1 0];
coefMat2 = [L^3 L^2 L 1; (2*L)^3 (2*L)^2 2*L 1; 3*L^2 2*L 1 0; ...
3*(2*L)^2 2*(2*L) 1 0];
coefMat3 = [(2*L)^3 (2*L)^2 (2*L) 1; (3*L)^3 (3*L)^2 (3*L) 1; ...
3*(2*L)^2 2*(2*L) 1 0; 3*(3*L)^2 2*(3*L) 1 0];

invcoefMat1 = inv(coefMat1);
invcoefMat2 = inv(coefMat2);
```

```
invcoefMat3 = inv(coefMat3);

figure(2);
for ii=2:nsteps
    d1 = Usol(1,ii);
    d2 = Usol(2,ii);
    d3 = Usol(3,ii);

    coef1 = invcoefMat1*[0; d1; 0; 0];
    coef2 = invcoefMat2*[d1; d2; 0; 0];
    coef3 = invcoefMat3*[d2; d3; 0; 0];

    yp1 = polyval(coef1, xp1);
    yp2 = polyval(coef2, xp2);
    yp3 = polyval(coef3, xp3);

    plot(yp1, xp1, 'b-', 'LineWidth', 2);
    hold on;
    plot(yp2, xp2, 'b-', 'LineWidth', 2);
    plot(yp3, xp3, 'b-', 'LineWidth', 2);
    plot(yp1+d, xp1, 'b-', 'LineWidth', 2);
    plot(yp2+d, xp2, 'b-', 'LineWidth', 2);
    plot(yp3+d, xp3, 'b-', 'LineWidth', 2);
    plot([d1 d+d1], [L, L], 'r-', 'LineWidth', 4);
    plot([d2 d+d2], [2*L, 2*L], 'r-', 'LineWidth', 4);
    plot([d3 d+d3], [3*L, 3*L], 'r-', 'LineWidth', 4);
    axis([-0.25 0.75 0.0 1.75]);
    text(0.6,1.6,['t = ' num2str(t(ii))]);
    hold off;
    pause(0.01);
end

% Compare with MATLAB ode45 solution:
[tmat, ymat] = ode45('strdynfun', [0 Tspan], [u0; v0]);
figure(1)
hold on;
pause;
plot(tmat,ymat(:,1),'r', tmat,ymat(:,2),'b', tmat,ymat(:,3),'k');
```