The Basics of MATLAB

Jeffrey O. Bauer
Contents

MATLAB Basics ........................................................................................................... 1

Typographical Conventions ........................................................................................ 1

Running MATLAB ......................................................................................................... 1

Starting MATLAB ......................................................................................................... 1
Evaluating Expressions ................................................................................................. 2
Variables ...................................................................................................................... 3
Functions and Commands ............................................................................................. 5
Saving Your Work and Loading External Data ......................................................... 6
MATLAB’s Workspace .................................................................................................. 7
MATLAB’s Help and Helpwin ...................................................................................... 8
MATLAB Preferences ................................................................................................. 9
Editing What You Have Typed .................................................................................... 10

Arrays and Matrices ................................................................................................. 10

Creating an Array (Vector) ......................................................................................... 10
Subscripts and Sub-Arrays ......................................................................................... 11
Array Arithmetic ......................................................................................................... 11
Deleting an Element ..................................................................................................... 13
Creating a Matrix .......................................................................................................... 13
Subscripts and Sub-Matrices ....................................................................................... 14
Matrix Arithmetic ......................................................................................................... 14
Solving a Linear Equation ............................................................................................ 17
Special Matrices .......................................................................................................... 19

Additional Variables .............................................................................................. 19

Complex Numbers ..................................................................................................... 19
Strings .......................................................................................................................... 21

Polynomials ............................................................................................................. 23

Plotting Polynomials ................................................................................................. 25

Plots ............................................................................................................................ 25

Line Plots ..................................................................................................................... 25
Annotating and Formatting Plots ............................................................................... 27
Subplots ....................................................................................................................... 29
Surface Plots ................................................................................................................ 30
Contour Plots ............................................................................................................... 32

Symbolic Algebra ...................................................................................................... 33
MATLAB Programming Basics ........................................... 39

Script Files ........................................................................ 39
  m-files ................................................................. 39
Side Effects .................................................................. 41
Comments .................................................................... 42

Function Files .................................................................. 42
  Syntax ........................................................................ 42
  Input and Output Arguments .................................... 43
  Primary and Secondary Functions ......................... 44

Input and Output ............................................................ 46
  User Input ............................................................. 46
  Text Output ............................................................ 46

Flow Control .................................................................... 50
  Relational and Logical Operators .................................. 50
  Operator Precedence ..................................................... 52
  if ... else ... Statements ............................................. 53
  switch Statements ...................................................... 54
  for Loops ................................................................... 55
  while Loops .............................................................. 56
  break ...................................................................... 57
  return ....................................................................... 58

Vectorization .................................................................. 58
  Vectors vs. Loops ......................................................... 58
  Copying ...................................................................... 58
  Pre-allocating Memory ................................................. 60

Four Elegant Devices ......................................................... 60
  Variable Input and Output ............................................. 60
  Global Variables ........................................................ 60
  feval ...................................................................... 61
  Inline Functions .......................................................... 61
## Numerical Problem Solving Basics

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Solving Methodology</td>
<td>63</td>
</tr>
<tr>
<td>Report of Solution</td>
<td>63</td>
</tr>
<tr>
<td>Work</td>
<td>63</td>
</tr>
<tr>
<td>Stepwise Refinement</td>
<td>63</td>
</tr>
<tr>
<td>Organizing and Documenting</td>
<td>64</td>
</tr>
<tr>
<td>Prologue</td>
<td>64</td>
</tr>
<tr>
<td>Visual Blocking</td>
<td>65</td>
</tr>
<tr>
<td>Meaningful Variable Names</td>
<td>66</td>
</tr>
<tr>
<td>Comments</td>
<td>66</td>
</tr>
<tr>
<td>Robust Programming</td>
<td>67</td>
</tr>
<tr>
<td>Problem Examples</td>
<td>67</td>
</tr>
<tr>
<td>Nonlinear Equation</td>
<td>68</td>
</tr>
<tr>
<td>Linear System</td>
<td>72</td>
</tr>
<tr>
<td>Curve Fitting</td>
<td>76</td>
</tr>
<tr>
<td>Numerical Integration</td>
<td>81</td>
</tr>
<tr>
<td>References</td>
<td>85</td>
</tr>
</tbody>
</table>
MATLAB Basics

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numerical computation. Using MATLAB, you can solve technical computing problems faster than with traditional programming languages, such as C, C++, and Fortran.

MATLAB contains mathematical, statistical, and engineering functions to support all common engineering and science operations. These functions, developed by experts in mathematics, are the foundation of the MATLAB language. The core math functions use the LAPACK and BLAS linear algebra subroutine libraries and the FFTW Discrete Fourier Transform library.

All the graphics features that are required to visualize engineering and scientific data are available in MATLAB. These include 2-D and 3-D plotting functions, 3-D volume visualization functions, tools for interactively creating plots, and the ability to export results to all popular graphics formats. You can customize plots by adding multiple axes; changing line colors and markers; adding annotation, LaTeX equations, and legends; and drawing shapes.

MATLAB provides a high-level language and development tools that let you quickly develop and analyze your algorithms and applications. The MATLAB language supports the vector and matrix operations that are fundamental to engineering and scientific problems. It enables fast development and execution. With the MATLAB language, you can program and develop algorithms faster than with traditional languages because you do not need to perform low-level administrative tasks, such as declaring variables, specifying data types, and allocating memory. In many cases, MATLAB eliminates the need for ‘for’ loops. As a result, one line of MATLAB code can often replace several lines of C or C++ code. At the same time, MATLAB provides all the features of a traditional programming language, including arithmetic operators, flow control, data structures, data types, object-oriented programming (OOP), and debugging features.

This document corresponds to MATLAB 2006R+ and the toolboxes added to it for Wayne State College, Nebraska.

Typographical Conventions

Throughout this guide commands will be presented in mono-spaced font, i.e. `>> help`.

Running MATLAB

Starting MATLAB

To invoke MATLAB open a terminal and type the shell command `matlab`. A MATLAB splash screen appears and a few seconds later MATLAB will open. The cursor should be setting behind the prompt “`>>`” in the command window. MATLAB is command driven for the most part. Menus are used mostly to manage files and the display.
The easiest way to learn how to use MATLAB is by doing. So, start playing. It is recommended that you try all the examples in this guide.

If you get into trouble, you can usually interrupt a MATLAB process by typing Control-C (usually written C-c for short). C-c gets its name from the fact that you type it by holding down CTRL and then pressing c. You will normally return the MATLAB prompt.

To exit MATLAB, type quit or exit at the prompt. You can also exit by clicking on the X in the upper right corner. (MATLAB functions like Windows-based programs.)

A great deal of work can be accomplished in the MATLAB command environment. However, there are other screens that can be accessed that can help productivity.

**Evaluating Expressions**

Expressions are evaluated by entering them after the >> followed by a carriage return. MATLAB displays the values preceded by “ans =”.

MATLAB follows the algebraic order of operations. Unary operators have right associativity and binary operators have left associativity.

**Unary Operators**
- + (positive)
- - (negative)

**Binary Operators**
- + (addition)
- - (subtraction)
- * (multiplication) (Multiplication is not inferred.)
- / (left-hand division)
- \ (right-hand division)
- ^ (exponent)
- ( ) (only grouping symbols allowed.)

```matlab
>> 2 + 5 - 3
ans = 4

>> ans - 3
ans = 1

>> a = 12, b = 5
a = 12
```
b = 
  5

>> c = sqrt(a^2 + b^2)

  c =
      13

>> 2/3

  ans =
       0.6667

>> 2\3

  ans =
       1.5000

>> pi

  ans =
       3.1416

>> sin(pi/3)

  ans =
       0.8660

>> log(10)

  ans =
       2.3026

**Variables**

MATLAB allows you to assign variables to numeric and/or string values. This is accomplished using the equals sign, “=” . There were a few examples of variable assignments in the previous section.

>> variable_name = 13

variable_name =
  13
>> array_name = [2 3 4 5]

array_name =
    2 3 4 5

>> string_name = 'string'

string_name =
    string

You can suppress output to the screen by ending an instruction or assignment with a “;”.

>> junk = 'jjjjjjjjjjjjjjjjuuuuuuuuuuuunnnnnnkkkkk';

The name of a variable can contain up to 31 characters. The name must start with a letter. The remaining 30 characters can be letters, numbers, or the underscore, “_”. Variable names are case sensitive, e.g. x and X are not the same.

Variable names should be descriptive but not so long they are unwieldy.

Be careful not to use the name of a built-in constant or function to name your variables. MATLAB doesn’t stop you from doing so. A strategy to determine if a name is already being used is to 1) type `help var_name_of_your_choice` and read the response, and 2) type `var_name_of_your_choice` and see if the name has been assigned.

**Built-in constants for MATLAB include the following:**
- **ans** (default variable name)
- **pi** (\(\pi = 3.14\ldots\))
- **Inf** (infinity = \(1/0\))
- **NaN** (not a number = \(0/0\))
- **i** or **j** (sqrt(-1))
- **eps** (floating point precision)
- **realmax** (largest positive floating point number)
- **realmin** (smallest positive floating point number)

**Built-in functions for MATLAB include the following:**
- **sqrt(expression)** (primary square root)
- **abs(expression)** (absolute value)
- **sin(expression)**
- **cos(expression)**
- **tan(expression)**
- **sec(expression)**
- **csc(expression)**
- **cot(expression)**
- **asin(expression)**
acos(expression)
atan(expression)
atan2(y, x)  (inverse tangent that is quadrant specific for the supplied y- and x-coordinates)
exp(expression)  (exponential function with base e = 2.718...)
log(expression)  (logarithmic function with base e = 2.718...)
log10(expression)  (logarithmic function with base 10)
log2(expression)  (logarithmic function with base 2)

Functions and Commands

Functions and commands are semantically different. Functions usually have input and output arguments. For example

>> y = sin(pi/3)

uses the sin function with the input argument of pi/3 and assigns the value of its output argument to y. The input arguments of functions are surrounded by parentheses.

A command such has help can have an input argument. But, the argument is not surrounded by parentheses. The command clear needs no arguments. Most commands stream output to the command window or perform some control function.

The truth be told, commands are functions, however most functions are not commands. This means that a command can be invoked with a parenthosized argument. Functions cannot be invoked with un-parenthesized arguments, i.e. log 10 will produce a meaningless result.

Built-in commands for MATLAB include the following:

- clc  (clears the command window)
- clear  (clears the workspace)
- who  (lists the variables in memory)
- ver  (displays the version of MATLAB and all toolboxes that are installed)
- help  (text-based help system)
- pwd  (displays the present working directory you are in)
- ls  (lists the files in the current directory)
- format  (formats the numerical output)
- save  (saves the workspace)
- load  (loads a saved workspace or data file)

The format command allows you to control the manner in which numeric values are displayed. The command does not affect the manner in which computations are performed. The default is short. Some of the choices are:
• format short  (Scaled fixed point format with 5 digits.)  
• format long (Scaled fixed point format with 15 digits for double and 7 digits for single.)  
• format short e  (Floating point format with 5 digits.)  
• format long e (Floating point format with 15 digits for double and 7 digits for single.)  
• format short g  (Best of fixed or floating point format with 5 digits.)  
• format long g (Best of fixed or floating point format with 15 digits for double and 7 digits for single.)

Saving Your Work and Loading External Data

Four main components of the MATLAB environment are 1) the command window, 2) the workspace, 3) graph windows, and 4) the editor. The command window is where you have been doing your work thus far. It displays the instructions that you have been using as well as the output to those instructions. In order to save the work you have been doing a diary must be opened. A diary is a text file that records the text that is displayed in the command window. No graphs are recorded.

DIARY

The diary command is used to open a pre-existing diary or to open a new one. Any text displayed prior to opening the diary is not recorded to the diary. If a diary already exists when opened, all text will be written to the end of the existing file.

>> diary mydiary.txt

or

>> diary('mydiary.txt')

A diary is closed with the diary off command or it is closed automatically when exiting MATLAB.

A saved diary can be opened and edited with any text editor or word processor.

WORKSPACE

The workspace consists of all the variables and their values that are used during a work session. It can be saved to a data file with a .mat suffix. The command save is used.

>> save my_variables

The file will be saved to the present working directory (pwd). You can save to a different directory by changing to the directory before issuing the save command, or by providing a
complete path name for the file. If you are saving to a subdirectory of the \texttt{pwd}, the name of the subdirectory must be included with the file name.

\input{code}

You can also save individual variables by name.

\input{code}

To save graphs or script files created with an editor it is easiest to use the save choice on the file menu.

You can load external data using the \texttt{load} command. The data must be stored as a \texttt{.mat} file in the MATLAB path.

\input{code}

If the data is not stored in the MATLAB path, then a complete path name must be used, similar to what was done with save.

\input{code}

\textbf{MATLAB's Workspace}

MATLAB's workspace is a graphical screen that is used to manage variables during a work session. A variable can be removed from memory by executing the command \texttt{clear variable_name}. \texttt{clear} used without a variable name will purge the entire memory. Functions as well as variables can be cleared.

The MATLAB workspace can also be managed graphically. Once in the workspace screen, a new variable can be added or an existing one, edited. Using the array editor is the most efficient way of entering very large matrices. For a 10x10 matrix that contains mostly zeros, execute the instruction

\input{code}
Now switch to the workspace and double click on the variable \texttt{a} (or select \texttt{a} and choose the edit option). A spreadsheet like screen will open. You can then select the appropriate cells and change their entries.

You can also use the workspace to delete variables.

The workspace also has a graphing feature. Different arrays can be selected and a host of different plots can be displayed without using the command screen.

If the workspace is not already opened, the command \texttt{workspace} will open it.

\textbf{MATLAB's Help and Helpwin}

MATLAB has two versions of help. Typing \texttt{help} in the command window will display a text list of the available functions and scripts. \texttt{help function_name} will display a synopsis of a function or script.

\begin{verbatim}
>> help abs
ABS    Absolute value.
   ABS(X) is the absolute value of the elements of X. When
   X is complex, ABS(X) is the complex modulus (magnitude) of
   the elements of X.
   See also \texttt{sign}, \texttt{angle}, \texttt{unwrap}.
   Overloaded functions or methods (ones with the same name in
   other directories)
          \texttt{help sym/abs.m}
Reference page in Help browser
          \texttt{doc abs}
\end{verbatim}
The command `lookfor` keyword will search for functions or scripts that use the word in their description.

```
>> lookfor inverse
INVHILB Inverse Hilbert matrix.
IPERMUTE Inverse permute array dimensions.
ACOS Inverse cosine.
ACOSD Inverse cosine, result in degrees.
ACOSH Inverse hyperbolic cosine.
ACOT Inverse cotangent.
ACOTD Inverse cotangent, result in degrees.
ACOTH Inverse hyperbolic cotangent.
ACSC Inverse cosecant.
ACSCD Inverse cosecant, result in degrees.
ACSCH Inverse hyperbolic cosecant.
ASEC Inverse secant.
ASECD Inverse secant, result in degrees.
ASECH Inverse hyperbolic secant.
ASIN Inverse sine.
ASIND Inverse sine, result in degrees.
ASINH Inverse hyperbolic sine.
ATAN Inverse tangent.
ATAN2 Four quadrant inverse tangent.
ATAND Inverse tangent, result in degrees.
ATANH Inverse hyperbolic tangent.
ERFCINV Inverse complementary error function.
ERFINV Inverse error function.
INV Matrix inverse.
PINV Pseudoinverse.
IFFT Inverse discrete Fourier transform.
IFFT2 Two-dimensional inverse discrete Fourier transform.
IFFTN N-dimensional inverse discrete Fourier transform.
IFFTSHIFT Inverse FFT shift.
```

The command `helpwin` works the same as `help`, but will display the information in a web browser that can be navigated by topic or index.

The command `doc` `topic` will launch the on-line documentation.

**MATLAB Preferences**

MATLAB can be configured for personal preferences. The preferences menu is under the File menu. Screen layout can be configured with the Desktop menu. The final layout can then be saved. To reduce start up time you should close unused screens, especially the helpwin screen. Before exiting MATLAB you should close all screens except the command window.
Editing What You Have Typed

At the MATLAB prompt, you can recall, edit, and reissue previous instructions using Emacs- or vi-style editing instructions. The default keybindings use Emacs-style instructions. For example, to recall the previous instruction, type Control-p (usually written C-p for short). C-p gets its name from the fact that you type it by holding down CTRL and then pressing p. Doing this will normally bring back the previous line of input. C-n will bring up the next line of input, C-b will move the cursor backward on the line, C-f will move the cursor forward on the line, etc.

You can also use the up- and down-arrow to navigate through previously issued instructions. If you retype or reissue an instruction, all the instructions that followed its initial issuance may also need to be reissued in their chronological order.

Arrays and Matrices

Creating an Array (Vector)

To create a new array (vector) and store it in a variable so that you can refer to it later, type the instruction

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

MATLAB will respond by printing the array in a neatly spaced row. MATLAB can also create column vectors, for example

```matlab
>> b = [1, 2, 3, 4]'
```

```
1
2
3
4
```

The " ' " is a transpose instruction that in this case converts a row vector into a column vector. Sequences can be produced as follows

```matlab
>> c = 1:100;
```

Ending an instruction with a semicolon prevents MATLAB from printing to the screen. The result of the above instruction would be the sequence 1, 2, 3, ..., 100 stored as an array.

```matlab
>> d = 1:5:100;
```

The 5 in the example above serves as a step size. The result is the sequence 1, 6, 11, ..., 96 stored as array d. If the step size is omitted (as in the first example) it is taken to be 1.
You can also count backwards.

```
>> e = 100:-2:0
```
i.e. 100, 98, 96, ... 0.

To create an array with equally spaced elements between two values do the following.

```
>> f = linspace(2, 5, 10)
```

```
f =
  2.0000  2.3333  2.6667  3.0000  3.3333  3.6667  4.0000  4.3333
      4.6667  5.0000
```
The last number (10 in the previous example) is the number of desired elements in the array.

**Subscripts and Sub-Arrays**

The elements of an array are identified with a subscript (index) inside of parentheses, e.g.

```
>> f(5)
```

```
an =
   3.3333
```
Subscripts must be a natural number, i.e. 1, 2, 3, 4, ...

A sub-array can be formed from a larger array by using the sequence notation introduced earlier.

```
>> sub_f = f(2:6)
```

```
sub_f =
  2.3333  2.6667  3.0000  3.3333  3.6667
```

**Array Arithmetic**

MATLAB has a convenient operator notation for performing array arithmetic. A “.” is placed in front of the arithmetic operators (already described). Arrays must be the same length.

```
>> a = 2:2:12; b=[3 5 7 11 13 17];
>> a.*b
```
```
an =
  6  20  42  88  130  204
```
>> a./b
ans =
    0.6667    0.8000    0.8571    0.7273    0.7692    0.7059

>> a.
ans =
    1.5000    1.2500    1.1667    1.3750    1.3000    1.4167

>> a.^2
ans =
     4    16    36    64   100   144

Most array operations are performed element by element. Functions can also be applied to elements of an array.

>> sqrt(b)
an =
     1.7321    2.2361    2.6458    3.3166    3.6056    4.1231

>> cos(b)
ans =
    -0.9900    0.2837    0.7539    0.0044    0.9074    -0.2752

The argument of a trigonometric function must be in radians.

A couple of useful array (vector) functions include:

- \texttt{sum(array)} (adds the elements of an array)
- \texttt{length(array)} (gives the number of elements in an array)
- \texttt{norm(vector)} (provides the Euclidean norm [magnitude] of a vector)
- \texttt{cross(vector\_1, vector\_2)} (produces the cross product of 2 vectors)
- \texttt{dot(vector\_1, vector\_2)} (produces the dot product of 2 vectors)

>> sum(a)
an =

    42

>> length(a)
an =

    6
>> norm(a)
ans =
    19.0788

>> u=[1 2 -3]; v=[-2 5 -10];
>> cross(u,v)
ans =
    -5    16     9

>> dot(u,v)
ans =
    38

**Deleting an Element**

Elements can be from arrays in the following fashion.

>> f(3)=[]
f =
    2.0000  2.3333  3.0000  3.3333  3.6667  4.0000  4.3333  4.6667
    5.0000

>> f(2:4)=[]
f =
    2.0000  3.6667  4.0000  4.3333  4.6667  5.0000

**Creating a Matrix**

To create a new matrix and store it in a variable so that it you can refer to it later, type the instruction

>> a = [1, 1, 2; 3, 5, 8; 13, 21, 34]

MATLAB will respond by printing the matrix in neatly aligned columns. Remember ending an instruction with a semicolon will prevent the result from being printed to the command window.

>> b = rand (3, 2);

will create a 3 row, 2 column matrix with each element set to a random value between zero and one. You can check in the workspace to see that a matrix was created.
To display the value of any variable, simply type the name of the variable. For example, to display the value stored in the matrix \( b \), type the instruction

\[
\begin{align*}
\text{>> } b \\
b & = \\
\begin{bmatrix}
0.9501 & 0.4860 \\
0.2311 & 0.8913 \\
0.6068 & 0.7621
\end{bmatrix}
\end{align*}
\]

**Subscripts and Sub-Matrices**

The elements of a matrix are identified with subscripts (indices) inside of parentheses, e.g.

\[
\begin{align*}
\text{>> } b(3,2) \\
\text{ans } &= \\
&= 0.7621
\end{align*}
\]

Subscripts must be natural numbers, i.e. 1, 2, 3, 4, \ldots The first index identifies the row and the second index identifies the column.

A sub-matrix can be formed from a larger matrix in the same fashion as was described for arrays.

\[
\begin{align*}
\text{>> } c &= a(1:2,2:3) \\
c & = \\
\begin{bmatrix}
1 & 2 \\
5 & 8
\end{bmatrix}
\end{align*}
\]

An entire row or column can be referenced with a "\( : \)". The instruction \( b(2,:) \) refers to row 2 of matrix \( b \).

\[
\begin{align*}
\text{>> } b(2,:) \\
\text{ans } &= \\
&= \begin{bmatrix}
0.2311 \\
0.8913
\end{bmatrix}
\end{align*}
\]

\( b(:,2) \) refers to column 2.

\[
\begin{align*}
\text{>> } b(:,2) \\
\text{ans } &= \\
&= \begin{bmatrix}
0.4860 \\
0.8913 \\
0.7621
\end{bmatrix}
\end{align*}
\]

**Matrix Arithmetic**

MATLAB uses the same arithmetic operators on matrices that it uses for evaluation. That is because MATLAB stores all values as matrices.
>> a=rand(3), b=rand(3)

a =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

b =
    0.4447    0.9218    0.4057
    0.6154    0.7382    0.9355
    0.7919    0.1763    0.9169

>> a*b  % inner dimensions must match
ans =
    1.0831    1.3151    1.2586
    0.6660    0.8743    0.9445
    1.3894    1.2668    1.7123

>> a+b  % matrices must be the same size
ans =
    1.3948    1.4078    0.8622
    0.8466    1.6295    0.9540
    1.3988    0.9384    1.7383

>> a^2  % a must be a square matrix
ans =
    1.2921    1.2428    0.8176
    0.4369    0.9208    0.1372
    1.2512    1.6002    0.9658

>> a'    % ''' is the transpose symbol
ans =
    0.9501    0.2311    0.6068
    0.4860    0.8913    0.7621
    0.4565    0.0185    0.8214

The symbol “%” is used to mark a comment. MATLAB will ignore anything typed after the % symbol (on the same line).
Most functions can be applied to matrices. Some functions that might be of interest include:

- `norm(matrix)` (provides the Euclidean norm [magnitude] of a matrix)
- `size(matrix)` (gives the number of rows and columns in a matrix)
- `[U L]=eig(matrix)` (produces the eigenvectors and eigenvalues of a matrix)
- `[L U]=lu(matrix)` (produces the LU factorization of a matrix)

**Example:**

```matlab
>> a
a =
0.9501 0.4860 0.4565
0.2311 0.8913 0.0185
0.6068 0.7621 0.8214

>> norm(a)
an =
1.8109

>> size(a)
an =
3 3

>> [L U]=eig(a)
L =
-0.6571 -0.7865 0.7614
-0.2275 0.3771 -0.6380
-0.7186 0.4892 0.1154

U =
1.6175 0 0
0 0.4332 0
0 0 0.6121

>> [L U]=lu(a)
L =
1.0000 0 0
0.2433 1.0000 0
0.6387 0.5843 1.0000
```
\[ U = \begin{bmatrix} 0.9501 & 0.4860 & 0.4565 \\ 0 & 0.7731 & -0.0925 \\ 0 & 0 & 0.5839 \end{bmatrix} \]

**Solving a Linear Equation**

Given a linear matrix equation of the form \( A x = b \), it can be easily solved with MATLAB’s \( \backslash \) - division. For example

\[
\begin{bmatrix} 2 & 3 \\ -5 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \end{bmatrix}
\]

\[
\text{>> } A=[2 \ 3; \ -5 \ 4] \\
A = \\
\begin{bmatrix} 2 & 3 \\ -5 & 4 \end{bmatrix} \\
\text{>> } b=[2 \ 12]' \\
b = \\
\begin{bmatrix} 2 \\ 12 \end{bmatrix} \\
\text{>> } x=A\backslash b \\
x = \\
\begin{bmatrix} -1.2174 \\ 1.4783 \end{bmatrix} \\
\text{>> } A*x \quad \text{checking the solution} \\
\text{ans =} \\
\begin{bmatrix} 2.0000 \\ 12.0000 \end{bmatrix}
\]

The process illustrated above is conceptually equivalent to \( x = \text{inv}(A) \ast b \), and works efficiently for small matrices, i.e. 3x3 and 4x4. It does not work if the matrix \( A \) is singular.

If \( A \) is singular (or nearly singular), you should use the \texttt{rref} function.
\texttt{\textgreater\textless{} A = [ 2 \ -3 \ 7; 12 \ 5 \ -13; -4 \ 6 \ -14]}
\begin{align*}
A = \\
\begin{bmatrix}
2 & -3 & 7 \\
12 & 5 & -13 \\
-4 & 6 & -14
\end{bmatrix}
\end{align*}

\texttt{\textgreater\textless{} b = [21 \ -15 \ 12]'}
\begin{align*}
b = \\
\begin{bmatrix}
21 \\
-15 \\
12
\end{bmatrix}
\end{align*}

\texttt{\textgreater\textless{} det(A) \% testing whether A is singular}
\begin{align*}
\text{ans} = \\
0
\end{align*}

\texttt{\textgreater\textless{} A \backslash b \% trying to solve with a singular matrix}

\textit{Warning: Matrix is close to singular or badly scaled.}
\begin{itemize}
\item Results may be inaccurate. \text{RCOND} = 1.502066e-017.
\end{itemize}
\begin{align*}
\text{ans} = \\
1.0e+016 \times \\
0.1322 \\
3.6347 \\
1.5200
\end{align*}

\texttt{\textgreater\textless{} rref([A \ b])}
\begin{align*}
\text{ans} = \\
\begin{bmatrix}
1.0000 & 0 & -0.0870 & 0 \\
0 & 1.0000 & -2.3913 & 0 \\
0 & 0 & 0 & 1.0000
\end{bmatrix}
\end{align*}

Therefore the solution is a line that can be described by the parametric set \textbf{C}: \textit{x}=0.0870t, \textit{y}=2.3913t, \textit{z}=t, \textit{t} is a Real number.
Special Matrices

MATLAB has several commands that will produce special matrices. Such commands include
- `ones()` (matrix of 1s)
- `zeros()` (matrix of 0s)
- `eye()` (identity)
- `inv()` (inverse)
- `det()` (result is not a matrix) (determinant)

```matlab
>> ones(3)
an =
  1 1 1
  1 1 1
  1 1 1
>> zeros(3,2)
an =
  0 0
  0 0
  0 0
```

Additional Variables

For most applications MATLAB variables will be assigned to numeric and/or string values. The numeric values will usually be real or complex. With the addition of the Symbolic toolbox, variables can also be declared as symbolic variables. This section will briefly describe some aspects of working with complex and string variables.

Complex Numbers

MATLAB has fully integrated complex arithmetic. All variables are considered to be complex and basic arithmetic operations perform correctly on both the real- and imaginary-part of complex numbers. The imaginary unit is automatically assigned to `i` and `j`. A complex number is entered as follows:

```matlab
>> z = 1 + 2i
z =
  1.0000 + 2.0000i
>> w = 1 - 2j
w =
  1.0000 - 2.0000i
```
The \( i \) and \( j \) must appear at the end of the expression. Be careful not to accidentally reassign \( i \) and \( j \) while working.

\[
\begin{align*}
\text{>> } & z+w \\
& \text{ans } = \\
& \phantom{=} 2 \\
\text{>> } & z-w \\
& \text{ans } = \\
& \phantom{=} 0 + 4.0000i \\
\text{>> } & z*w \\
& \text{ans } = \\
& \phantom{=} 5 \\
\text{>> } & z/w \\
& \text{ans } = \\
& \phantom{=} -0.6000 + 0.8000i
\end{align*}
\]

MATLAB’s \texttt{exp} function supports Euler’s notation for complex numbers \( (z = a + bi = re^{i\theta}) \). In Euler’s notation \( r \) is the magnitude or absolute value of the number and \( \theta \) is the angle formed with the polar (real) axis.

\[
\begin{align*}
\text{>> } & r = 5; \theta = 53.13\pi/180; \\
& z = r\exp(i\theta) \\
& z = 3.0000 + 4.0000i \\
\text{>> } & \text{abs}(z) \\
& \text{ans } = \\
& \phantom{=} 5 \\
\text{>> } & \text{angle}(z)\times 180/\pi \\
& \text{ans } = \\
& \phantom{=} 53.1300
\end{align*}
\]

Three other useful functions for working with complex numbers are \texttt{real}, \texttt{imag} and \texttt{conj}. 

- 20 -
>> real(z), imag(z), conj(z)
ans =
    3.0000
ans =
    4.0000
ans =
    3.0000 - 4.0000i

**Strings**

Strings are interpreted as arrays with character elements. String constants are enclosed in single quotes.

>> first_name = 'Jonathon';
>> last_name = 'Smith';
>> middle_initial='P';
>> name = [first_name, ' ', middle_initial, '. ', last_name]
nname =

    Jonathon P. Smith

The last line above concatenates the three string variables with spacing and a period, and assigns it to a new variable.

>> length(name)
ans =
    17

>> first = name(1:8)
first =

    Jonathon

>> last = name(13:17)
last =

    Smith
Since strings are interpreted as arrays of characters, you can use some of the array functions previously described on them (as shown above).

Strings can be transposed.

```matlab
>> last'
ans =
Smith
```

Several strings can be stored in a matrix. But the strings must be the same length since each will serve as a row of the matrix. To avoid adding additional spaces, you can use the `str2mat` function.

```matlab
>> name_matrix=str2mat(first_name,middle_initial,last_name)
name_matrix =

Jonathon
P
Smith
```

```matlab
>> size(name_matrix)
an =

3     8
```

A numeric value can be converted to a string with the `num2str` function. This conversion will allow additional formatting with the `sprintf` function. The `sprintf` function parameters are very similar to those used in the C language.

```matlab
>> format long
>> sqrt3=sqrt(3)
sqrt3 =

1.73205080756888
```

```matlab
>> sqrt3_str=num2str(sqrt3) % num2str will only display 4 places
sqrt3_str =

1.7321
```
>> sqrt3_str=num2str(sqrt3,20) % the 20 allows for 20 characters
sqrt3_str =
1.7320508075688772

>> sprintf('The principal square root of 3 is %20s.',sqrt3_str)
an =
The principal square root of 3 is 1.7320508075688772.

Polynomials

MATLAB has several different functions for working with polynomials. Polynomials are entered in as arrays of coefficients. (Note: insert 0s for missing terms.) The function roots() will locate the zeros of the polynomial.

>> p = [1 2 3 4];
>> roots(p)
an =
-1.6506
-0.1747 + 1.5469i
-0.1747 - 1.5469i

A polynomial can also be built from an array of roots using the poly() function.

>> P=poly(ans)
P =
1.0000 2.0000 3.0000 4.0000

You can evaluate a polynomial at a given value or list of values by using polyval(), for example

>> polynomial = [1 2 -3 4];
>> xvalues = 1:5;
>> yvalues = polyval(polynomial, xvalues)
yvalues =
4 6 10 16 24

Two polynomials can be convoluted (multiplied) using conv(p, q), or deconvoluted (divided) using deconv(p, q).
>> Q=[2 -3 4];
>> P_Q=conv(P,Q)

\[ P_Q = \]
\[
\begin{bmatrix}
2.0000 & 1.0000 & 4.0000 & 7.0000 & -0.0000 & 16.0000 \\
\end{bmatrix}
\]

>> P=[2 4 -5 6];
>> d=[2 6];
>> [Q R]=deconv(P,d)

\[ Q = \]
\[
\begin{bmatrix}
1.0000 & -1.0000 & 0.5000 \\
\end{bmatrix}
\]

\[ R = \]
\[
\begin{bmatrix}
0 & 0 & 0 & 3 \\
\end{bmatrix}
\]

The quotient is \( x^2 - x + 0.5 \) and the remainder is 3.

Partial fractions for a rational expression can be obtained using the `residue()` instruction, for example

\[
\frac{5x + 3}{x^2 + 7x + 10}
\]

can be decomposed by the following instructions:

>> n=[5 3];d=[1 7 10];
>> [r s]=residue(n,d)

\[ r = \]
\[
\begin{bmatrix}
7.3333 \\
-2.3333 \\
\end{bmatrix}
\]

\[ s = \]
\[
\begin{bmatrix}
-5 \\
-2 \\
\end{bmatrix}
\]

Thus

\[
\frac{5x + 3}{x^2 + 7x + 10} = \frac{22}{3} \frac{1}{(x + 5)} + \frac{1}{3} \frac{1}{(x + 2)}
\]
Plotting Polynomials

MATLAB uses a simple plot technique where a set of domain values can be plotted against a set of range values. The two sets do not necessarily create a function. Polynomials can be plotted in the following manner:

```matlab
>> polynomial = [1 2 -3 4];
>> xvalues = 1:15;
>> yvalues = polyval(polynomial, xvalues);
>> plot(xvalues, yvalues)
```

Refer to the next section for details on formatting the plot.

Plots

MATLAB is a very powerful visualization tool that can produce many 2-dimensional and 3-dimensional plots, as well as animation. You can produce publication quality plots with the formatting tools.

The data for plotting can be stored in vectors, matrices, and external data files, or defined by analytic functions.

Line Plots

A basic 2-D plot is a line plot of one variable versus another. Below are a few examples.

```matlab
>> x = -2*pi:pi/12:2*pi;
>> y=sin(x);
```
>> plot(x,y)
Annotating and Formatting Plots

There are several formatting features that can be used to dress up a plot. These include:

- `title(string)`
- `xlabel(string)`
- `ylabel(string)`

```matlab
>> x=rand(1,10);
>> y=rand(1,10);
>> % produces a scatter plot using o to mark the points
>> plot(x,y,'o')
```

![Scatter plot with points marked by 'o'](image-url)
• `legend(string 1, string 2, ...)`
• `text(x, y, string)`
• `axis([x_min x_max y_min y_max])`
• `grid` (choose either on or off)

```matlab
>> x = -2*pi:pi/12:2*pi;
>> y=sin(x);
>> y2=cos(x);
>> plot(x,y,x,y2,'--')
>> title('sin(\theta) & cos(\theta)')
>> xlabel('\theta in radians')
>> legend('sin(\theta)','cos(\theta')
>> grid
```

To learn more about the plotting functions listed above perform a `help` on each. Over 75 symbols can be included in text strings. This can be accomplished by embedding a subset of \TeX commands in the string. There is also a limited set of \TeX formatting commands available. They include `^` and `_` for superscripts and subscripts, respectively. Also included are

• `\fontname`
• `\fontsize`
• `\bf` (boldface)
• `\it` (italic)
• `\sl` (slant)
• `\rm` (Roman)
Subplots

You can place more than one plot on a figure using the `subplot` function. The function takes three arguments.

```matlab
subplot(n_rows, n_cols, which_plot)
```

`n_rows` and `n_cols` serve to divide the figure into fields. The fields are numbered from left to right and then up to down. `which_plot` tells which field to place a plot.

```matlab
>> x=linspace(0,10,100);
>> y1=log(x);y2=log(2*x);
Warning: Log of zero.
Warning: Log of zero.
>> y3=log(3*x);y4=log(4*x);
Warning: Log of zero.
Warning: Log of zero.
>> subplot(2,2,1)
>> plot(x,y1), title('y=ln(x)')
>> subplot(2,2,2)
>> plot(x,y2), title('y=ln(2x)')
>> subplot(2,2,3)
>> plot(x,y3), title('y=ln(3x)')
>> subplot(2,2,4)
>> plot(x,y4), title('y=ln(4x)')
```

![Subplots](image-url)
Surface Plots

There are several choices for plotting \( z = f(x, y) \). The most common ways are to use mesh and surf. When defining a function of two independent variables, you need to first set up a rectangular domain. You use meshgrid for this.

\[
[x \ y]=\text{meshgrid}(-10: 10); \quad \% \text{both } x \text{ and } y \text{ will range between } -10 \text{ to } 10
\]

Now you can define \( z \) as a function of \( x \) and \( y \).

\[
> z = 3*x.^2 + y.^2; \quad \% \text{remember to use array operators}
\]

Finally a 3-D plot can be made using either mesh or surf.

\[
> \text{mesh}(x, y, z); \\
> \text{xlabel}('x-axis'); \text{ylabel}('y-axis'); \text{zlabel}('z-axis'); \\
> \text{grid on}
\]
You can change the viewing angles for a 3-D plot with `view(azimuth, elevation)`. The default angles are -37.5° and 15°. The azimuth is taken from the positive x-axis and the elevation is from the xy-plane.

`>> view(45,30)`
**Contour Plots**

Level curves of a surface can be produced with the `contour` function. It has several different forms that include:

**CONTOUR** Contour plot.
- `CONTOUR(Z)` is a contour plot of matrix Z treating the values in Z as heights above a plane. A contour plot are the level curves of Z for some values V. The values V are chosen automatically.
- `CONTOUR(X,Y,Z)` X and Y specify the (x,y) coordinates of the surface as for SURF.
- `CONTOUR(Z,N)` and `CONTOUR(X,Y,Z,N)` draw N contour lines, overriding the automatic value.
- `CONTOUR(Z,V)` and `CONTOUR(X,Y,Z,V)` draw LENGTH(V) contour lines at the values specified in vector V. Use `CONTOUR(Z,[v v])` or `CONTOUR(X,Y,Z,[v v])` to compute a single contour at the level v.

The above description was obtained by using

```
>> help contour
```

```
>> contour(x,y,z,[0:50:350])
```

Contour plots can be combined with surface plots by appending a c on the end of `mesh` or `surf`.

```
>> surfc(x,y,z); grid on
```
Symbolic Algebra

MATLAB uses the Maple™ kernel to perform symbolic algebra.

If you wish to perform symbolic algebra, you must first declare the variables you are going to use in expressions as symbolic variables. This is done by using the `sym` command.

```
>> sym x t y
```

Expressions containing the symbolic variables will be considered a symbolic object that can be manipulated.

Below are a few examples of how you do symbolic algebra.

```
>> syms x
>> f=x^3+3*x^2+3*x+1
f =
   x^3+3*x^2+3*x+1
>> pretty(f)
   3      2
   x  + 3 x  + 3 x + 1
```
>> F=factor(f)
F =

(x+1)^3

>> collect(F)
ans =
x^3+3*x^2+3*x+1

>> simple(f)
simplify:
x^3+3*x^2+3*x+1
radsimp:
x^3+3*x^2+3*x+1
combine(trig):
x^3+3*x^2+3*x+1
factor:
(x+1)^3
expand:
x^3+3*x^2+3*x+1
combine:
x^3+3*x^2+3*x+1
convert(exp):
x^3+3*x^2+3*x+1
convert(sincos):
x^3+3*x^2+3*x+1
convert(tan):
x^3+3*x^2+3*x+1
collect(x):
\[x^3+3*x^2+3*x+1\]
mwcos2sin:
\[x^3+3*x^2+3*x+1\]
ans =
\[(x+1)^3\]

>> F
F =
\[(x+1)^3\]

>> expand(F)
ans =
\[x^3+3*x^2+3*x+1\]

>> g=F/(x^2-2*x+1)
g =
\[(x+1)^3/(x^2-2*x+1)\]

>> pretty(g)
\[\frac{(x + 1)^3}{x^2 - 2 \, x + 1}\]

>> pretty(factor(g))
\[\frac{(x + 1)^3}{2 \, (x - 1)}\]

>> expand(g)
ans =
\[1/(x^2-2*x+1)*x^3+3/(x^2-2*x+1)*x^2+3/(x^2-2*x+1)*x+1/(x^2-2*x+1)\]
\[
\text{pretty(expand(g))}
\]
\[
\frac{3}{x} + \frac{2}{x-2x+1} + \frac{3}{x-2x+1} + \frac{1}{x-2x+1}
\]
\[
\text{pretty(simplify(g))}
\]
\[
\frac{3}{x+1}
\]
\[
\text{limit(g,-1)}
\]
\[
\text{ans = 0}
\]
\[
\text{limit(g,1)}
\]
\[
\text{ans = Inf}
\]
\[
\text{gp=diff(g) \% finds the derivative in terms of x}
\]
\[
gp =
3*(x+1)^2/(x^2-2x+1) - (x+1)^3/(x^2-2x+1)^2*(2x-2)
\]
\[
\text{pretty(gp)}
\]
\[
\frac{2}{3} \frac{(x + 1)}{x - 2x + 1} - \frac{3}{2} \frac{(x + 1)(2x - 2)}{(x - 2x + 1)^2}
\]
>> pretty(simplify(gp))

\[
\frac{2}{(x + 1) (x - 5)} - \frac{2}{(x - 1) (x - 2 x + 1)}
\]

>> syms y
gp = sin(x*y)
h = sin(x*y)

>> hx = diff(h,x) \% finds the derivative in terms of x
hx = cos(x*y)*y

>> hxy = diff(hx,y) \% finds the derivative in terms of y
hxy = -sin(x*y)*x*y+cos(x*y)

>> hx = int(hxy,y) \% finds the integral of hxy in terms of y
hx = -1/x*(sin(x*y)-x*y*cos(x*y))+1/x*sin(x*y)

>> simplify(hx)
as = cos(x*y)*y

You can plot a symbolic object in one variable using the instruction ezplot.

>> hx = simplify(hx);
>> ezplot(hx)
>> ezplot(hx, [0 10])
MATLAB Programming Basics

MATLAB integrates a high-level programming language into an interactive environment that lets you quickly develop and analyze algorithms. The MATLAB language’s main data type is a matrix. Vectors and matrices are integral to scientific computing and problem solving making MATLAB an excellent choice for such tasks. Use of vectors and matrices can also greatly reduce the amount of code necessary to solve scientific problems. Less code also means less runtime.

The language is reminiscent of the Fortran-language with a touch of C-language. MATLAB is an interpretive language. That is to say there is no compiling. Most low-level tasks (i.e. declaring variables and allocating memory) are unnecessary.

Script Files

There is no doubt that you can do a lot of work in the command window. However, when the amount of instructions needed to complete a task increases or you need to re-execute a block of instructions several times, the command window is a poor choice. The best choice is to write a MATLAB program.

MATLAB programs are stored as text files. The files can be loaded into the MATLAB command environment where they are then interpreted and executed. MATLAB supports two types of programs: script files and function files. Script files consist of sequences of instructions stored in text files. The files are commonly referred to as m-files because of the .m extension used.

m-files

m-files consist of lists of MATLAB instructions stored as a text file. An m-file can be created by any text editor or word processor. If a word processor is used, make sure to save the file as a text file. MATLAB has its own built-in text editor. To launch it, type the command edit in the command window.
The use of the editor is pretty straight-forward. The advantage of using MATLAB’s editor is that color coding and formatting is built in. Built-in commands and functions appear as blue, comments appear as green and strings show up as purple. Flow structures are automatically indented.

Another advantage of using an m-file is that m-files have access to all variables in the current workspace and all variables created by m-files are stored in the workspace. You could use an m-file to set constants that will be needed during a work session. Suppose that you are working on fluid mechanics and the ideal gas law \( p = \rho RT \), where \( R \) is the gas constant. You could write a script that would set \( R \) for common gases.

\[ \text{>> edit gas_constants} \]

The command above will launch the MATLAB editor and open the file gas_constants if it exists. If the file does not exist, MATLAB will ask you if you would like to create it.

\[
\begin{align*}
\% \text{Gas Constants (J/kg*K)} \\
\% \text{Common gases at Standard Atmospheric Pressure (SI units)} \\
\% \text{Temperature in degrees Celsius} \\
\% \text{Temperature for all gases is 20 except for air which is 15} \\
\% \\
R_{\text{air}} &= 2.869e+2; \\
R_{\text{cd}} &= 1.889e+2; \\
R_{\text{he}} &= 2.077e+3; \\
R_{\text{h}} &= 4.124e+3; \\
R_{\text{me}} &= 5.183e+2; \\
R_{\text{n}} &= 2.968e+2; \\
R_{\text{o}} &= 2.598e+2;
\end{align*}
\]

Once the m-file is created and saved, you execute it by typing the name of the script (without the .m extension) in the command window.

\[ \text{>> gas_constants} \]

Type the command \texttt{who} to see that the constant have been set.

\[ \text{>> who} \]

Your variables are:

\[
R_{\text{air}} \hspace{5pt} R_{\text{cd}} \hspace{5pt} R_{\text{h}} \hspace{5pt} R_{\text{he}} \hspace{5pt} R_{\text{me}} \hspace{5pt} R_{\text{n}} \hspace{5pt} R_{\text{o}}
\]
% Sine & Cosine Plot
%
% The following script plots both the y = \sin(x) and y = \cos(x)
% over the range from -2*\pi and 2*\pi. The graph is formatted and
% a legend is displayed.

x = -2*pi:pi/12:2*pi;
y = sin(x);
y2 = cos(x);
plot(x,y,x,y2,'--')
title('\textit{sin(\theta) \& ... cos(\theta)}')
xlabel('\textit{\theta in radians}')
legend('\textit{sin(\theta)}', ' \textit{cos(\theta)}')
axis([-2*pi 2*pi -1.5 1.5])
grid

\begin{center}
\begin{tikzpicture}
\begin{axis}[
    xlabel={\textit{\theta in radians}},
    ylabel={sin(\theta) \& cos(\theta)},
    grid=both
    ]
    \addplot[blue, thick] coordinates {(-2*pi,0) (2*pi,0)};
    \addplot[green, dashed, thick] coordinates {(-2*pi,0) (2*pi,0)};
\end{axis}
\end{tikzpicture}
\end{center}

\textbf{Side Effects}

The advantage of being able to set variables using an m-file is also a major drawback. If you already have variables in the workspace with the same names as those in a script, the script will replace the values. Perhaps that is what you want to have happen, usually it isn’t.

In the sine and cosine plot above, the variables \( x \), \( y \) and \( y2 \) were added to the workspace and could very easily overwritten pre-existing values of those variables.
Comments

The two scripts above made use of comment statements. Comments consist of the characters between the % symbol and the end of the line. The % symbol can appear anywhere within an instruction line. Comments were made on several examples in the first chapter of this manual. Multiple lines of comments must each begin with a % symbol.

The % can also be used to “hide” programming statements from the interpreter. This is a common practice when debugging code.

It is very important to fully document any program that you code. This will be brought up in the next chapter.

Function Files

A function file is a hybrid script file that also uses a .m extension. A function file (or simply a function) is analogous to a Fortran subroutine or a C function. Functions are code modules that can communicate with the MATLAB command window and each other. A function utilizes a predefined list of input and output arguments. A MATLAB function is easier to work with than functions in other languages (e.g. C-language). A MATLAB function can output more than one matrix (remember that the matrix is MATLAB’s data type) where as in a C program you can output only one value and would have to use pointers to manipulate the elements of a matrix.

Variables that are used in a function are considered local and are invisible to other functions and the command window. If you want a function to be used by another function or the command window you can include it as an output argument. The values of output arguments are sent to the workspace. Another option is to use a global variable. Care should be taken when using global variables. Global variables should not be used when input and output arguments can accomplish the same task. No further discussion of global variables will be made.

Syntax

The main syntax issue of a function is the definition line. The definition line of a function must be the first instruction line and follow the following form

\[
\text{function } [\text{output}_1, \text{output}_2, \ldots] = \text{function}_\text{name}(\text{input}_1, \text{input}_2, \ldots)
\]

where the output list is comma separated and enclosed by brackets ([ ]), and the input list is comma separated and enclosed in parentheses ( ). Both lists are optional. The function name must follow the variable conventions described in the first chapter. You should save a function under the name used in the definition line.

A function with no input or output list is referred to as a null function. Null functions can be used in lieu of m-files without fear of inadvertently changing variable values.
%% Cosine & Sine Plot Function
%%
%% The following script plots both the y = sin(x) and y = cos(x)
%% over the range from -2*pi and 2*pi. The graph is formatted
%% and a legend is displayed.

function cosine_sine_plot

x = -2*pi:pi/12:2*pi;
y=sin(x);
y2=cos(x);
plot(x,y,x,y2,'--')
title('\fontname{times} \fontsize{16} \sin(\theta) & \cos(\theta)')
xlabel('\fontname{times} \fontsize{16} \theta in radians')
legend('\fontname{times} \fontsize{10} \sin(\theta)', ...'
      '\fontname{times} \fontsize{10} \cos(\theta)')
axis([-2*pi 2*pi -1.5 1.5])
grid

>> clear
>> who
>> cosine_sine_plot
>> who
>>

**Input and Output Arguments**

As mentioned previously the input and output lists can be of any length. The arguments in either list can be numeric- or a string-valued. In most applications you will know what arguments you want to feed to a function and which arguments you want a function to return. You can also call a function with fewer input or output arguments than listed. nargin and nargout are variables available to a function that stores the number of arguments fed to the function and the number of output arguments being requested. Both variables are used to help make a function robust.

In some cases you may not know the number of arguments to feed a function or to send back to the command window. You can use vargin and varargout in those cases. Both are arrays that store the input and output arguments. You can access each argument with an index, i.e. vargin(1).

You can also include a function as an input argument. The name of the function is passed as a string.
Primary and Secondary Functions

The functions as so far described are primary functions and are accessible to any other function and/or the command window. A secondary or sub-function can be included with a primary function and will be seen only by the primary function. Secondary functions appear at the end of a primary function.

%% truss: Isosceles Triangular Truss
%% Solve for the members of a simple isosceles triangular truss with vertical point loads.
%% usage f=truss(b, theta, P)
%% Input:
%% b - length of the truss
%% theta - base angle of the truss in degrees
%% P - vector containing the point load on the pins of the truss starting with the left pin
%% Output:
%% f - vector containing the forces in the three members of the truss [left strut, bottom chord, right strut]
%% tension is positive
%% compression is negative

%% By: Jeffrey O. Bauer
%% Date: 01/03/07

%% ---------------- Primary Function ----------------
function f=truss(b, theta, P)

theta=theta*pi/180; % convert the angle to radians
R = reactions(b,P); % find the reactions
[A b]=matrices(theta, R, P); % determine the matrices
f=inv(A)*b; % solve for the forces
% Find the reactions of a simply supported truss
% Input:
% b - length of the truss
% P - vector containing the point load on the
% pin of the truss starting with the left
% pin
% Output:
% R - vector containing the two reactions
% left to right

function R=reactions(b,P)

R=[0 0]';
M=P(2)*(b/2)+P(3)*b;
R(2)= M/b;
R(1)= sum(P)-R(2);

% Create the matrices for the matrix equation
% Input:
% theta - base angle of the truss in degrees
% R - vector containing the two reactions
% left to right
% P - vector containing the point load on the
% pins of the truss starting with the left
% pin
% Output:
% A - coefficient matrix
% b - constant vector

function [A b]=matrices(theta, R, P)

c=cos(theta); s=sin(theta);
A=[c 0 ;s 0; -c 0 c];
b=[0 0 0]';
b(2)=P(1)-R(1);
>> P=100*ones(3,1);
>> f = truss(12, 30, P)
f =

-100.0000
100.0000
-100.0000

**Input and Output**

In most situations input and output is streamed through a function by the input and output argument list of the function. There are occasions when it is necessary to prompt a user for information or display results in a formatted fashion.

**User Input**

The `input` function is used to prompt users for information. The input data can be either numeric- or string-valued and is entered via the keyboard. The function expects a numeric-valued argument.

>> x=input('prompt ')
prompt

MATLAB will display the prompt and wait for a value or matrix to be entered. If you want a string (such as a name of a function) you must include a second argument which is `'s'`.

>> x=input('string prompt ', 's')
string prompt

**Text Output**

There are three main functions for printing output. The first is the low-level function `disp`. It will display the value of a variable or a string.

>> format long
>> disp(pi)
3.14159265358979

>> disp('This is a string!!!')
This is a string!!!

The advantage of using display is that it does not show the variable name on the screen. Draw back is that the display formatting is handled by the `format` command, which is limited.

*This example and many others in this chapter are shown as command window instructions in order to demonstrate their results. All MATLAB instructions can be part of an m-file or function.*
Two functions that are available in MATLAB are `sprintf` and `fprintf`. Both functions can be used to print formatted output to the screen. The first function `sprintf` is intended to convert data to a string.

```matlab
>> phi = sprintf('%0.15g',(1+sqrt(5))/2)
phi =
1.61803398874989
>> size(phi) % this demonstrates that the value is a string
ans =
    1   16
>> sprintf('%0.5g', eps)
an=
2.2204e-016
>> sprintf('%1.25f', eps)
an=
0.00000000000000002220446049
>> sprintf('%15.5g', 1/eps)
an=  
    4.5036e+015
>> sprintf('%15.5f', 1/eps)
an=  
    4503599627370496.00000
```

The formatting (conversion) specifications used are similar to those used in the C-language. Conversion specifications are pefaced with the character % and involve optional flags, optional width and precision fields, optional subtype specifier, and conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to a C-language manual for complete details. The special formats \n, \r, \t, \b, and \f can be used to produce linefeed, carriage return, tab, backspace, and formfeed characters respectively. Use \\ to produce a backslash character and %% to produce the percent character.

Remember that your data is now stored as a string. If you need to convert back to numeric-valued data, use the `str2num` function. The function takes the string and converts it to a matrix.
of numeric values. The string you wish to convert must consist of only numeric characters. MATLAB uses the `eval` function to interpret strings.

```matlab
>> str='(1+sqrt(5))/2'
str =
(1+sqrt(5))/2
>> eval(str)
ans =
 1.61803398874989
```

```matlab
>> str='clc % this command clears the command screen'
str =
clc % this command clears the command screen
```

The last function is `fprintf`. This function formats data and streams it to a file. If you omit the file name as an argument, the data is streamed to the command window. A noticeable difference between `sprintf` and `fprintf` is that `fprintf` does not assign and display a variable name.

```matlab
>> fprintf('string');
string
>> fprintf('%0.15g',(1+sqrt(5))/2)
1.61803398874989
>> fprintf('%0.5g', eps)
2.2204e-016
>> fprintf('%1.25f', eps)
0.00000000000000002220446049
>> fprintf('%15.5g', 1/eps)
4.5036e+015

>> fprintf('%15.5f', 1/eps)
```
The same formatting specifications used with `sprintf` are used. Refer to the previous description.

Previously it was mentioned that the filename must be omitted in order to print to the command window. The example below shows how to print formatted text to a file named `data.dat`.

```matlab
>> x=[1:10]';
>> fout=fopen('data.dat','wt');
>> fprintf(fout,'   k     x(k)\n');
>> for k=1:length(x)
    fprintf(fout,'%4d    %5.2f\n',k,x(k));
end
>> fclose(fout);
```

You can see the contents of any `.m`, or `.dat` with the `type` command. To view an m-file you do not need to include the `.m` extension.

```matlab
>> type polyder
% this is a MATLAB provided function

function [a,b] = polyder(u,v)
    %POLYDER Differentiate polynomial.
    % POLYDER(P) returns the derivative of the polynomial whose
    % coefficients are the elements of vector P.
    %
    % POLYDER(A,B) returns the derivative of polynomial A*B.
    %
    % [Q,D] = POLYDER(B,A) returns the derivative of the
    % polynomial ratio B/A, represented as Q/D.
    %
    % Class support for inputs u, v:
    %    float: double, single
    %
    % See also POLYINT, CONV, DECONV.

    % Copyright 1984-2004 The MathWorks, Inc.
    % $Revision: 5.11.4.2 $ $Date: 2004/03/02 21:47:56 $

if nargin < 2, v = 1; end

u = u(:).'; v = v(:).';
u = length(u); nv = length(v);
if nu < 2, up = 0; else up = u(1:nu-1) .* (nu-1:-1:1); end
if nv < 2, vp = 0; else vp = v(1:nv-1) .* (nv-1:-1:1); end
a1 = conv(up,v); a2 = conv(u,vp);
i = length(a1); j = length(a2); z = zeros(1,abs(i-j));
if i > j, a2 = [z a2]; elseif i < j, a1 = [z a1]; end
```
if nargout < 2, a = a1 + a2; else a = a1 - a2; end
f = find(a ~= 0);
if isempty(f), a = a(f(1):end); else a = zeros(superiorfloat(u,v)); end
b = conv(v,v);
f = find(b ~= 0);
if isempty(f), b = b(f(1):end); else b = zeros(class(v)); end

Below is the result of writing to data.dat in the previous example.

>> type data.dat

<table>
<thead>
<tr>
<th>k</th>
<th>x(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td>4</td>
<td>4.00</td>
</tr>
<tr>
<td>5</td>
<td>5.00</td>
</tr>
<tr>
<td>6</td>
<td>6.00</td>
</tr>
<tr>
<td>7</td>
<td>7.00</td>
</tr>
<tr>
<td>8</td>
<td>8.00</td>
</tr>
<tr>
<td>9</td>
<td>9.00</td>
</tr>
<tr>
<td>10</td>
<td>10.00</td>
</tr>
</tbody>
</table>

**Flow Control**

A great majority of algorithms require conditional execution of blocks of code and/or iterative processes applied to blocks of code. The MATLAB language provides the same basic flow control as other high-level languages.

**Relational and Logical Operators**

Relational and logical operators are used to form logical statements that can be determined true or false. True statements are assigned a value of 1 and false statements are assigned a value of 0.

>> a=5; b=25; c=5;
>> is_a_smaller_than_b = (a<b)
is_a_smaller_than_b =

1

>> is_a_less_than_or_equal_to_b = (a <= b)
_a_less_than_or_equal_to_b =

1
Some of the relational and logical operators available are listed in the table below.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Relational, binary</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Relational, binary</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Relational, binary</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Relational, binary</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>==</td>
<td>Relational, binary</td>
<td>Equal</td>
</tr>
<tr>
<td>~=</td>
<td>Relational, binary</td>
<td>Not equal</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical, binary</td>
<td>And</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical, binary</td>
</tr>
<tr>
<td>~</td>
<td>Logical, unary</td>
<td>Not</td>
</tr>
</tbody>
</table>

All non-zero real-values are considered true and 0 is the only value considered false. Complex numbers, NaN, and Inf cannot be converted to a truth value of 0 or 1.

A logical statement can be used as an array index as shown below.

```matlab
>> a=[-1 2 -3 4 5 -6];
>> a(a>0) % the positive elements are sorted out
ans =
  2 4 5
```

Some useful logical functions are included in the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>isvector(variable)</td>
<td>Is true if the variable represents a vector</td>
</tr>
<tr>
<td>isscalar(variable)</td>
<td>Is true if the variable represents a single element array</td>
</tr>
<tr>
<td>isnumeric(variable)</td>
<td>Is true if all elements are numeric</td>
</tr>
<tr>
<td>isempty(variable)</td>
<td>Is true if variable has been assigned to [ ] (the empty set)</td>
</tr>
<tr>
<td>ischar(variable)</td>
<td>Is true if the variable represents a string character</td>
</tr>
<tr>
<td>isnan(variable)</td>
<td>Is true if the variable has been assigned to not a number, e.g. 0/0</td>
</tr>
<tr>
<td>isinf(variable)</td>
<td>Is true if the variable has been assigned to infinity, e.g. 1/0</td>
</tr>
<tr>
<td>isfinite(variable)</td>
<td>Is true if the variable represents a finite value</td>
</tr>
</tbody>
</table>

```matlab
>> A=[2 3 4 5];
>> isnumeric(A)
```
ans =
   1

>> isvector(A)
an =
   1

>> isvector([])
an =
   0

>> isempty([])
an =
   1

>> a=[-2:2];b=[1 -1 0 -2 2];
>> c=a./b
Warning: Divide by zero.
c =
   2.0000  1.0000  NaN  -0.5000  1.0000

>> isnan(c)
an =
   0  0  1  0  0

**Operator Precedence**

There are three types of operators, 1) arithmetic, 2) relational, and 3) logical. The order of precedence for the operators is the same as the list above, i.e. arithmetic operators have the highest precedence. The order of arithmetic operations was discussed in the first chapter. A more complete hierarchy is shown below. Operators of the same precedence are grouped together between bold lines.
Consider the following examples.

\[
\begin{align*}
y &= 3 > 8 - 2 \mid \sqrt{15} > 4 \\
y &= 3 > 6 \mid 3.8730 > 4 \\
y &= 0 \mid 0 \\
y &= 0
\end{align*}
\]

\[
\begin{align*}
y &= (8^2 - 5 \mid 8 - 2 < 3) \& \neg(\sqrt{12} > 5) \\
y &= (16 - 5 \mid 8 - 2 < 3) \& \neg(3.4641 > 5) \\
y &= (11 \mid 6 < 3) \& \neg(3.4641 > 5) \\
y &= (11 \mid 0) \& \neg0 \\
y &= 1 \& \neg0 = 1 \& 1 = 1
\end{align*}
\]

**if ... else ... Statements**

The main MATLAB decision construct is the if ... else ... statement. There are several variations of it. The simplest has the form

\[
\text{if (logical statement)} \\
\text{block of code to execute if logical statement is true} \\
\text{end}
\]
Other possible forms are

```plaintext
if (logical statement)
    block of code to execute if logical statement is true
else
    block of code to execute if logical statement is false
end
```

```plaintext
if (logical statement 1)
    block of code to execute if logical statement 1 is true
elseif (logical statement 2)
    block of code to execute if logical statement 2 is true
    ...
    ...
elseif (logical statement n)
    block of code to execute if logical statement n is true
else
    block of code to execute if all logical statements are false
end
```

**switch Statements**

An alternative to a long and complicated `else ... elseif ...` construct is the `switch` construct. Its general format is as follows.

```plaintext
switch evaluative expression
    case value_1
        block of code to execute if the evaluative expression is value_1
    case value_2
        block of code to execute if the evaluative expression is value_2
        ...
```
case value_n
    block of code to execute if the evaluative expression
    is value_n
otherwise
    block of code to execute if the evaluative expression
    is not any of the values
end

**for-Loops**

Iteration is prevalent in many programs and numeric routines. One of the most popular iteration constructs is a *for*-loop. The loop utilizes an index that typically counts from a starting value to an ending value and is used in a block of statements. The block is executed for each index value. Recursive definitions are natural examples for using a *for*-loop.

```matlab
function f=fibonacci(N)

if nargin < 1 | ischar(N), N=10; end
N=ceil(N);

f=ones(N,1);

f(1)=1;
f(2)=1;

for index=3:N
    f(index)=f(index-2)+f(index-1);
end
```

```matlab
%% fibonacci: Fibonacci sequence
%%
%% Generate the Fibonacci sequence
%%
%% usage f=fibonacci(N {should be a positive integer})
%%
%% Input:
%% N - the number of terms in the sequence
%%      should be a positive integer
%%
%% Output:
%% f - an array containing the sequence

%% By:     Jeffrey O. Bauer
%% Date:   01/04/07
```

- 55 -
>> f=fibonacci(4.3)
f =
1
1
2
3
5

**while-Loops**

The second most commonly used loop construct is the while-loop. This loop continues to execute a block of instruction until a condition is met. In several algorithms for solving nonlinear equations a block of instructions continues to be executed until a predetermined tolerance is met.

```matlab
%% fixed: Fixed Point Solution
%% Determine the solution of an equation expressed as
%% x=g(x) to within a given tolerance (error)
%% usage x=fixed(equation, x1, tolerance)
%% Input:
%% equation - String containing the equation to be solved
%% x1 - Initial guess at solution
%% tolerance - Allowable error (default is 0.001)
%% Outut:
%% x - Array containing progressively better guess at the solution. The last entry is the final solution.
%% By: Jeffrey O. Bauer
%% Date: 01/04/07

function x=fixed(eq, x1, tol)

len_eq=length(eq);

if eq(1)~='x'
    error('The equation must begin with x =');
end

if nargin<3
    tol=0.001;
```
if nargin <2
    x1=0;
end

n=1;
while eq(n)~='='
    n=n+1;
end
n=n+1;
equation = eq(n:len_eq);
n=1; x=x1; X(n)=x;
X(n+1)=eval(equation);

while abs(X(n+1)-X(n))>tol
    n=n+1; x=X(n);
    X(n+1)=eval(equation);
end

x=X';

**break**

When using a *while*-loop it is necessary to provide a “break”. Without a “break,” the stopping condition may never be reached creating an infinite loop. The `break` command stops execution of a loop and transfers the interpreter to the first instruction line directly after the `end` of the loop.

The `break` command can also be used when two or more stopping conditions are necessary. For example the previous fixed routine could be modified to incorporate a maximum number of iterations as a stopping mechanism. This is very common in routines of this type.

max_n=25;

while n<max_n-1
    n=n+1; x=X(n);
    X(n+1)=eval(equation);
    if abs(X(n)-X(n-1))<=tol
        break;
    end
end
**return**

The `return` command is very similar to the `break` command in the sense that it stops execution. However, the `return` command transfers control back to the calling function or command window, whereas the `break` command transfers control to the first instruction directly after the loop in the function it appears. Any code that follows the execution of a `return` command is not interpreted.

**Vectorization**

Vectorization refers to the process of transforming code that operates on scalars to code that operates on vectors. This type transformation makes code more efficient and the operations will be performed more quickly by MATLAB. Vector operations are performed using optimized binary code that is part of the MATLAB kernel.

**Vectors vs. Loops**

Vectors should be used instead of loops when loops contain many scalar operations. Below is a loop structure very similar to how it would be done in Fortran.

```matlab
k=1;       % initialize the counter
P=[1 -2 5]; % p(x) = x^2-2x+5
for x=0:0.1:10 % domain
    X(k)=x;
    Y(k)=polyval(P,x);
    k=k+1;
end
```

An alternative to the code above is

```matlab
P=[1 -2 5]; % p(x) = x^2-2x+5
x=0:0.1:10; % domain
y=polyval(P,x);
```

The MATLAB code is more concise and will execute quicker.

**Copying**

Vectorization makes copying elements from one vector or matrix to another more efficient as well. A common need when solving a system of linear equations iteratively is to create the matrices L, D, and U for a square matrix A such that L + D + U = A.

```matlab
%% decomp_mat: Decompose A Square Matrix
%%
%% Decompose (break down) a square matrix A
```
function [L D U]=decomp_mat(A)

[m n]=size(A);
if m ~= n
    error('The matrix A must be square!');
end

I=eye(m);
D=A.*I;
L=zeros(m);

for k = 2:m
    L(k,1:k-1)=A(k,1:k-1);
end

U=A-D-L;

>> A=rand(4)
A =
    0.9355  0.0579  0.1389  0.2722
    0.9169  0.3529  0.2028  0.1988
    0.4103  0.8132  0.1987  0.0153
    0.8936  0.0099  0.6038  0.7468

>> [L D U]=decomp_mat(A)
L =
     0       0       0       0
     0.9169  0.0000  0.0000  0.0000
     0.4103  0.8132  0.0000  0.0000
     0.8936  0.0099  0.6038  0.0000
\[ D = \]

\[
\begin{pmatrix}
0.9355 & 0.0000 & 0.0000 & 0.0000 \\
0.0000 & 0.3529 & 0.0000 & 0.0000 \\
0.0000 & 0.0000 & 0.1987 & 0.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.7468
\end{pmatrix}
\]

\[ U = \]

\[
\begin{pmatrix}
0 & 0 & 0 & 0 \\
-0.9169 & 0 & 0 & 0 \\
-0.4103 & -0.8132 & 0 & 0 \\
-0.8936 & -0.0099 & -0.6038 & 0
\end{pmatrix}
\]

The \texttt{decomp_mat} routine does make use of a \texttt{for}-loop, but it also uses vectorization to copy rows from the given matrix \( A \) to the matrix \( L \). \( D \) and \( U \) are found through matrix operations.

### Pre-allocation Memory

You do not need to pre-allocate memory since a MATLAB matrix is a dynamic data structure. Although, initializing a matrix or vector (especially inside a loop) makes code perform more efficiently. The functions \texttt{ones}, \texttt{zeros}, and \texttt{eye} are used for initializing a matrix. The \texttt{zeros} function was used in the \texttt{decomp_mat} routine.

### Four Elegant Devices

MATLAB has four devices that can help make programming more robust and elegant. They are variable input and output arguments, global variables, \texttt{feval}, and inline functions.

#### Variable Input and Output

Variable input and output arguments make functions more flexible and easier to use. The plot function is a prime example of use of variable input arguments. It accepts an unlimited number of vectors and formatting strings. Two very important variables make this possible: \texttt{nargin} and \texttt{nargout}. They contain the number of input arguments and output arguments that are being passed or requested when a function call is made. Their use is splattered in a few of the examples in this chapter.

#### Global Variables

Global variables are used to forgo with input and output arguments. A global variable is seen by all functions and can subsequently be manipulated by all functions. Global variables should not be used when input and output arguments can accomplish the needed task.
A variable can be made global by a declaration statement (global variable_name) in the command window or any function. The declaration must appear in each function that wishes to use the variable.

**feval**

Many applications involve user-defined functions that need to be evaluated. The functions are usually single- or multi-variable functions. The feval function is used to evaluate these.

```matlab
%% funn: Oxygen Diffusion around a Capillary
%%
%% Input:
%% r - R, K, B1, B2 - are constants determined by
%% the geometry, reaction rates and other specifics
%% of the problem
%%
%% Output:
%% c - Oxygen diffusion

function c=funn(r)

R=8;K=1;
B1=3;B2=1;

c1=(R.*r.^2)./(4*K);
c2=B1.*log(r);  % log is the natural log

c=c1+c2+B2;

>> r=[1:5]';
>> clear c
>> c=feval('funn',r)
c =
     3.0000
     11.0794
    22.2958
    37.1589
    55.8283
```

**Inline Functions**

If a function definition is not too long or complicated, it can be defined as an inline function. Inline functions are evaluated by passing values to them directly.
funn = inline('(cosh(x)).^2+(sinh(x)).^2;')

    Inline function:
    funn(x) = (cosh(x)).^2+(sinh(x)).^2;

funn(1)

ans =

    3.7622

x = -2:2;
funn(x)

ans =

    27.3082         3.7622         1.0000         3.7622         27.3082
Numerical Problem Solving Basics

Numerical problem solving (or scientific computing) involves a marriage of application, computation and mathematics. It is a form of scientific investigation done by formulating mathematical models whose solutions are approximated by computer simulations. A consistent and orderly approach to the formulation of models is important. The approach should be general enough to be applicable to all fields of study. An engineering methodology will be used.

Problem Solving Methodology

A common methodology for problem solving is described by six steps:

1. Clearly define the problem.
2. Describe the input and output data.
3. Work a hand example with simpler data.
4. Develop a computer model.
5. Test model against hand example.
6. Implement model and report solution.

Report of Solution

The report of the solution to a problem should be organized. A common organization of the report contains five parts:

1. Problem Statement
2. Find Statement
3. Given Statement
4. Work
5. Results

Work

Work involves both hand calculations and computer calculations. Computer calculations consist of work done in the MATLAB command window as well as the development of a MATLAB function (or functions).

Stepwise Refinement

When developing mathematical model for a problem or when developing a MATLAB function, you should perform what is known as stepwise refinement. Stepwise refinement involves breaking down a larger task into smaller tasks. Each smaller task is then further broken down into smaller modules that have obvious solutions or can be easily tested in the MATLAB command window. The smaller modules are then solved or coded. If they are coded then they and incorporated as part of a MATLAB function.
You should use a consistent style when developing a function.

**Organizing and Documenting a Function**

A consistent organization and documentation of a function is critical. It is critical in the sense that it helps assure a workable function. It also minimizes the number of bugs in the function and makes debugging easier. You can get a good sense of how to organize and document your functions by looking at how MATLAB functions are organized and documented. They are separated into five areas.

<table>
<thead>
<tr>
<th>function declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prologue</td>
</tr>
<tr>
<td>Process input arguments</td>
</tr>
<tr>
<td>Computational tasks</td>
</tr>
<tr>
<td>Prepare output arguments</td>
</tr>
</tbody>
</table>

The location of the function declaration can be either at the top or directly below the prologue.

**Prologue**

The prologue is the documentation of the function. The prologue should closely follow MATLAB conventions. An example of a function prologue is shown below.

```matlab
%% decomp_mat: Decompose A Square Matrix
%%
%% Decompose (break down) a square matrix A
%% such that A = L + D + U
%%
%% usage: [L D U]=decomp_mat(A)
%%
%% Input:
%% A - Square matrix
%%
%% Output:
%% L - Lower triangular matrix
```
%% D -        Diagonal matrix
%% U -        Upper triangular matrix

%% By:     Jeffrey O. Bauer
%% Date:   01/04/07

The main part of the prologue consists of 1) the function name, 2) a synopsis of the function, 3) usage of the function, 4) input arguments, and 5) output arguments. MATLAB uses the first unbroken comment block of a function in its help system. Therefore it is important to keep the main portion of the prologue together. Use % for spacing.

The second portion of the prologue is the credits. It is separated from the main portion by a space. It should contain information about the programmer and date of the original code. If the function is revised at a later date, the date of the revision should be included in the credits. The credits are not displayed when using MATLAB’s help.

>> help decomp_mat

% decomp_mat: Decompose A Square Matrix
% Decompose (break down)a square matrix A
% such that A = L + D + U
% usage [L D U]=decomp_mat(A)
% Input:
% A -       Square matrix
% Output:
% L -       Lower triangular matrix
% D -       Diagonal matrix
% U -       Upper triangular matrix

**Visual Blocking**

Visual consistency is an important part of a programming style. MATLAB’s text editor helps with this. It will automatically indent blocks in flow control constructs. It also color codes different aspects of the code.

You can also insert white spaces in your code. White spaces visually groups related lines of code together.
<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_n=25;</td>
<td>max_n=25;</td>
</tr>
<tr>
<td>while n &lt; max_n-1</td>
<td>while n&lt;max_n-1</td>
</tr>
<tr>
<td>n = n+1;</td>
<td>n=n+1;x=X(n);</td>
</tr>
<tr>
<td>x = X(n);</td>
<td>X(n+1)=eval(equation);</td>
</tr>
<tr>
<td>X(n+1)=eval(equation);</td>
<td>if abs(X(n)-X(n-1))&lt;=tol</td>
</tr>
<tr>
<td>if abs(X(n)-X(n-1))&lt;=tol</td>
<td>break;</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Note: visually appealing crappy code is still crappy code.

**Meaningful Variable Names**

A variable name can be up to 31 characters. The name must start with a letter. The remaining 30 characters can be letters, numbers, or the underscore, “_”. Variable names are case sensitive, e.g. `max_iterations` is not equal to `max_Iterations`. Descriptive variable names help in reading code and debugging. Avoid using generic names such as `x`, `y`, and `z`. Also avoid using too long of names they will become very cumbersome.

**Comments**

Use comments within the code sections to help break it apart and clarify the code’s purpose.

```matlab
function f=fibonacci(N)

% Processing the input
%
% Guaranteeing a positive integer argument

if nargin < 1 | ischar(N), N=10; end

N=abs(N);
N=ceil(N);

% Initializing the sequence

f=ones(N,1);

% Setting the first and second term of the sequence

f(1)=1;
f(2)=1;
```
% Recursive definition for the rest of the sequence

for index=3:N
    f(index)=f(index-2)+f(index-1);
end

Robust Programming

Robust programming means that you develop code to handle user ignorance. For example, the user entered a fraction when an integer is expected. Another example, a function requires three input arguments but the user only provides two. When developing a MATLAB function you should do the following:

- Never assume that input data are correct. Check that the proper type of data is inputted. Check to see that all the needed data is supplied.
- Guard against the occurrence of conditions that could prevent correct calculations.
- Use a default condition for if ... elseif ... else and switch constructs.
- Use error messages to help determine why a function failed.

When the error function is encountered a message is displayed and control is returned to the command window or calling function.

[m n] = size(A);

if m ~= n
    error('The matrix A must be square!');
end

Problem Examples

Four examples of solved problems follow. The problems are typical for beginning study in numerical problem solving (scientific computing).

- Nonlinear Equations
- Systems of Linear Equations
- Curve Fitting and Function Approximations
- Differentiation and Integration
Nonlinear Equation

Problem: Find the roots to a nonlinear equation

Find: Roots $x_1, x_2, \ldots$

Given: $f(x) = 0$

Work:

$x^2 = 3 \equiv x^2 - 3 = 0$
**Algorithm**

Graph equation and establish guesses at roots
Determine \( g(x) \)
Pass \( g(x) \) and \( x_1 \) to function
Set tolerance and maximum number of iterations
Set \( x(1) = x_1 \)
Repeat
  \[ x(n+1) = g(x(n)) \]
until \(|x(n+1) - x(n)| < \text{tolerance or iterations exceed maximum number of iterations}| \)
Computer Routine

%% fixed: Fixed Point Solution
%%
%% Determine the solution of an equation expressed as
%% x=g(x) to within a given tolerance (error)
%%
%% usage x=fixed(funn, x1, tolerance)
%%
%% Input:
%% funn - String containing the name of function
%% containing g(x)
%% x1 - Initial guess at solution
%% tolerance - Allowable error (default is 0.001)
%%
%% Output:
%% x - Array containing progressively better
%% guess at the solution. the last entry
%% is the final solution.

%% By: Jeffrey O. Bauer
%% Date: 01/05/07

function x = fixed(funn, x1, tol)

n=1;X(n)= x1; max_n=25;

%%%%%%%%%%%%%%%% Preparing Input %%%%%%%%%%%%%%%%%%%
if nargin < 3
    tol=0.001;
end
if nargin < 2
    x1=0;
end
if nargin < 1
    error('usage: x = fixed(funn, x1, tolerance)');
end

%%%%%%%%%%%%%%%% Main Loop %%%%%%%%%%%%%%%%%%%%%%%% 

while n < max_n
    X(n+1)=feval(funn,X(n));
    if abs(X(n+1)-X(n))<=tol
        break;
    end
    n=n+1;
end

x=X';
Test Routine

$$g = \text{inline}'0.5*(x+3./x)'$$

$$g =$$

\hspace{1cm} \text{Inline function:} \\
\hspace{1cm} g(x) = 0.5*(x+3./x)

$$x =$$

\begin{align*}
1.00000000000000 \\
2.00000000000000 \\
1.75000000000000 \\
1.73214285714286 \\
1.73205081001473 \\
\end{align*}

$$x =$$

\begin{align*}
-1.00000000000000 \\
-2.00000000000000 \\
-1.75000000000000 \\
-1.73214285714286 \\
-1.73205081001473 \\
\end{align*}

Results

The solution to the problem is the actual computer routine
Problem: Solve the linear system

\[ 4x + y = 6 \]
\[ -x + 5y = 9 \]

Find: Point of intersection

Given: Two equations above

Work: \[ 4x + y = 6 \rightarrow x = \frac{6 - y}{4} \]
\[ -x + 5y = 9 \rightarrow y = \frac{x + 9}{5} \]
Algorithm

Graph equations and establish guess at \( x_1 \)
Determine \( y=g(x) \) and \( x=h(y) \) as \( \text{funn} \)
Pass \( \text{funn} \) and \( x_1 \) to routine
Set tolerance and maximum number of iterations
Set \( x(1) = x_1 \)
Repeat
\[
[x(n+1) \ y(n)] = \text{funn}(x(n))
\]
until \( ||[x(n) \ y(n)] - [x(n-1) \ y(n-1)]|| < \text{tolerance} \) or
iterations exceed maximum number of iterations

Computer Routine

%%% fixedsys: Fixed Point Solution for 2x2 System
%%% Determine the solution to a 2x2 system of
%%% linear equations to within a given tolerance (error)
%%% usage \( X=\text{fixedsys}(\text{funn}, x_1, \text{tolerance}) \)
%%% Input:
%%% \( \text{funn} \) – String containing the name of function
%%% containing system
%%% \( x_1 \) – Initial guess at solution
%%% \( \text{tolerance} \) – Allowable error (default is 0.001)
%%% Output:
%%% \( X \) – Matrix containing progressively better
%%% guesses at the solution. The last row
%%% is the approximate solution.
%%% By: Jeffrey O. Bauer
function X = fixedsys(funn, x1, tol)

max_n=25;

%%%%%%%%%%% Preparing Input %%%%%%%%%%

if nargin < 3
    tol=0.001;
end

if nargin < 2
    x1=0;
end

if nargin < 1
    error('usage: X=fixedsys(funn, x1, tolerance)');
end

%%%%%%%%%%% Main Loop %%%%%%%%%%%%%%%%%%%%%

x(1)=x1;
[x(2) y(1)]=feval(funn,x(1));
n=2;

while n < max_n
    [x(n+1) y(n)]=feval(funn,x(n));
    if norm([x(n) y(n)]-[x(n-1) y(n-1)]) <= tol
        break;
    end
    n=n+1;
end

x=x(1:length(x)-1);
X=[x' y'];

Test Routine

function [x y]=funn(x)

y=(x+9)/5;
x=(6-y)/4;
>> X=fixedsys('funn',0,0.0001)

X =

       0   1.80000000000000  
  1.05000000000000   2.01000000000000  
  0.99750000000000   1.99950000000000  
  1.00012500000000   2.00002500000000  
  0.99993750000000   1.99998750000000  
  1.00000031250000   2.00000006250000

Results

The solution is approximately (1.00000031250000, 2.00000006250000)

Note: The routine developed for this system of linear equations would work for other 2x2 systems.
**Curve Fitting**

Problem: Find the function approximation for a population growth that has a limit of \( L = 5000 \).

Find: \( P(t) \) \& \( P(5) \)

Given:

<table>
<thead>
<tr>
<th>( t )</th>
<th>( P(t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>1800</td>
</tr>
<tr>
<td>3</td>
<td>2800</td>
</tr>
<tr>
<td>4</td>
<td>3700</td>
</tr>
</tbody>
</table>

\[
P(t) = \frac{L}{1 + Ce^{At}}
\]

Work:

\[
Y = \frac{L}{1 + Ce^{At}} \Rightarrow 1 + Ce^{At} = \frac{L}{Y}
\]

\[
\log(1 + e)
\]

\[
\log(\frac{L}{Y} - 1 = Ce^{At})
\]

\[
\log(\frac{L}{Y} - 1) = \log C + A\cdot t
\]

Let \( v = \log(\frac{L}{Y} - 1) \), \( u = t \), \( B = \log C \)

\[
\therefore v = Au + B
\]

\[
\begin{bmatrix}
\Sigma u^2 & \Sigma u v \\
\Sigma u v & \Sigma v
\end{bmatrix}
\begin{bmatrix}
A \\
B
\end{bmatrix}
= 
\begin{bmatrix}
\Sigma u v \\
\Sigma v
\end{bmatrix}
\]
Hand work in command window

>> P=[500 1000 1800 2800 3700]';
>> t=[0:4]';
>> plot(t,P,'o',[0 10],[5000 5000])
>> axis([0 10 0 5500]),grid

>> u=t;
>> v=log(5000./P-1);
>> M=[sum(u.^2) sum(u);sum(u) length(u)]
M =
    30    10
    10     5

>> C=[sum(u.*v) sum(v)]'
C =
     -2.3703
      2.8718

>> p=inv(M)*C
p =
     -0.8114
      2.1971

>> PP=inline('5000./(1+8.9991*exp(-0.8114*t))')
\[ PP(t) = \frac{5000}{1 + 8.9991 \exp(-0.8114t)} \]

\[
\text{Computed Routine}
\]

\[
\text{function population}
\]

\[
\% \text{ form of population model is }
\]

\[
\% \quad P(t) = \frac{L}{1 + C \exp(A t)}
\]

\[
L = 5000; \% \text{ set limit }
\]

\[
t = [0:4]'; \% \text{ set time }
\]

\[
\text{pop} = [500 \ 1000 \ 1800 \ 2800 \ 3700]'; \% \text{ set populations}
\]

\[
\text{disp}([t \ \text{pop}]) \% \text{ quick display of time vs. population}
\]

\[
\% \text{ linearize equation to } y = A t + B
\]

\[
u = t; \% \text{ transform independent variable}
\]

\[
v = \log(L./\text{pop} - 1); \% \text{ transform dependent variable}
\]

\[
\% \text{ solve } M \times X = b \text{ where } X = [A \ B]'
\]

\[
su = \text{sum}(u);
\]

\[
M = [\text{sum}(u.^2) \ \text{su}; \ \text{su} \ \text{length}(u)];
\]

\[
b = [\text{sum}(u.*v) \ \text{sum}(v)]';
\]

\[
X = \text{inv}(M) \times b;
\]

\[
A = X(1);
\]

\[
B = X(2);
\]

\[
C = \exp(B);
\]

\[
\% \text{ Find } P(5)
\]

\[
P_5 = \text{feval('P',A,C,L,5)}
\]

\[
\% \text{ Plot data vs. approximation}
\]

\[
x = 0:0.5:10;
\]
\texttt{y=feval('P',A,C,L,x);}  
\texttt{plot(t,pop,'o',x,y);}  
\texttt{title('Population Growth')}  
\texttt{xlabel('time')}  
\texttt{legend('Data','Approximation')}  

\texttt{%%%%%%%%%%%%%%%% Secondary Function %%%%%%%%%%%%%%}  
\texttt{function p=P(A,C,L,t)}  
\texttt{p=L./(1+C.*exp(A.*t));}  

\textbf{Test Routine}  
\texttt{>> population}  
\begin{verbatim}  
0 500  
1 1000  
2 1800  
3 2800  
4 3700  
\end{verbatim}  
\texttt{P_5 =}  
\texttt{4.3264e+003}
Results

The population model is
\[ P(t) = \frac{5000}{1 + 8.991e^{-0.8114t}} \]

Refer to graph above

\[ P(5) = 4.3264e+003 \]

Note: You could easily change the data at the start of the routine to suit another situation. You could also make the routine a function that is passed \( t, y, \) and \( L \).
Problem: Approximate the definite integral

\[ I = \int_{1}^{3} \frac{1}{x^3} \, dx \]

Find: \( I \)

Given: \( f(x) = \frac{1}{x^3}, a = 1, b = 3 \)

Work:

In general

\[
\int_{a}^{b} f(x) \, dx \approx \int_{a}^{b} \left[ f(a) + \frac{f(b)-f(a)}{h} (x-a) \right] \, dx
\]

\[ P_2(x) \quad \text{(Taylor Poly.)} \]

\[ h = (b-a) \]

\[ = \left[ f(a) x + \frac{f(b)-f(a)}{2h} (x-a)^2 \right]_{a}^{b} \]

\[ = \left[ f(a) h + \frac{(f(b)-f(a))h}{2} \right] \]

\[ = \frac{h}{2} \left[ f(a) + f(b) \right] \]

\[ I = \frac{2}{2} \left[ \frac{1}{1^3} + \frac{1}{3^3} \right] = \left[ 1 + \frac{1}{27} \right] = \frac{28}{27} \]
Hand example in command window

```matlab
>> f=inline('1./x.^3')
f =

    Inline function:
    f(x) = 1./x.^3

>> plot(x,f(x))
Warning: Divide by zero.
> In inlineeval at 13
    In inline.subsref at 25
>> axis([0 4 0 10])
>> axis([0 4 0 3])
>> axis([0 4 0 1.5])
>> grid
```

```
>> a=1
a =

    1

>> b=3
b =

    3
```

- 82 -
>> h=b-a
h =
2

>> I=(h/2)*(f(a)+f(b))
I =
1.0370

>> 28/27
ans =
1.0370

Computer Routine

function dIntegral

% set parameters
f=inline('1./x.^3'); % define function (integrand)
a=1; % lower limit
b=3; % upper limit
h=b-a; % step size

% plot and format graph
x=0:0.1:4;
plot(x,f(x));
axis([0 4 0 1.5])
grid

% display result
format rat % set display to rational numbers
I1=(h/2)*(f(a)+f(b)) % calculate I (one panel)

% suppose two panels
c=a+(b-a)/2;
h=h/2;
x=[a c b];
I2=(h/2)*sum((f(x(1:2))+f(2:3)))
Test Routine

>> dIntegral
Warning: Divide by zero.
> In inlineeval at 13
    In inline.subsref at 25
    In dIntegral at 13

I1 =

    28/27

I2 =

    139/216

Results

The approximation of \( \int_{1}^{3} \frac{1}{x^3} dx \) is 28/27 when one panel is used and 139/216 when two panels are used.
References


