Functions of Two Variables

- MATLAB allows us to work with functions of more than one variable
- With MATLAB 5 we can even move beyond the traditional $M \times N$ matrix to matrices with an arbitrary number of dimensions
- Functions of two variables or 2-D slices of $N$-dimensional functions are often of interest in engineering analysis
- Engineers in particular like to visualize functions of two variables using various types of three-dimensional (3-D) plots
- To create a function of two variables, $z = f(x, y)$, in MATLAB we need to form a grid of the underlying $x$ and $y$ values

Each point in the $x$-$y$ grid corresponds to a point in the $x$-$y$ plane where we evaluate $f(x, y)$

Two matrices, in general, are required to perform the evaluation:
- One whose columns have constant values reflecting changes in the $x$ variables, and
One whose rows have constant values reflecting changes in the y variable

- To obtain a matrix of the functional values $z = f(x, y)$ we load these $x$ and $y$ grid matrices into $f(x, y)$ and obtain a new matrix $z$ of the corresponding functional values

- A MATLAB function that creates grid matrices from $x$ and $y$ range vectors is `meshgrid(x, y)`
  ```matlab
  » x = -1:.1:1; % x-axis range vector
  » y = 0:.1:2; % y-axis range vector
  » [x_grid, y_grid] = meshgrid(x, y);
  » whos
  Name            Size         Bytes  Class
  x               1x21         168   double array
  x_grid         21x21        3528  double array
  y               1x21         168   double array
  y_grid         21x21        3528  double array
  
- Consider the function
  $$z = f(x, y) = \frac{1}{1 + x^2 + y^2}$$
• To generate the functional values matrix \( z \) we use the grid matrices as substitutions for \( x \) and \( y \) respectively

\[
\begin{align*}
\text{» } z &= \frac{1}{1 + x\text{\_grid} \cdot 2 + y\text{\_grid} \cdot 2}; \\
\text{» } \text{size}(z) \\
\text{ans} &= \\
21 & 21
\end{align*}
\]

3-D Plotting Options

• In MATLAB there are a variety of standard 3-D plotting functions; a short table is given below

<table>
<thead>
<tr>
<th>Plot Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{mesh}(x,y,z)</td>
<td>Plots a mesh type surface defined by matrix ( z ). ( x ) and ( y ) are either vectors of the ( x ) and ( y ) range values or the corresponding grid matrices.</td>
</tr>
<tr>
<td>\text{surf}(x,y,z)</td>
<td>Plots a shaded surface defined by matrix ( z ). ( x ) and ( y ) are as defined above.</td>
</tr>
<tr>
<td>\text{contour}(x,y,z)</td>
<td>A flat map containing contour lines of the surface defined by ( z ). ( x ) and ( y ) are as defined above. The number of contour lines is chosen automatically.</td>
</tr>
<tr>
<td>\text{contour}(x,y,z,v)</td>
<td>Same as three argument contour except ( v ) defines the contour lines.</td>
</tr>
<tr>
<td>\text{meshc}(x,y,z)</td>
<td>A mesh plot with a contour plot sitting in the ( z = 0 ) plane.</td>
</tr>
</tbody>
</table>
Example: Evaluate the diffraction pattern of a square aperture

\[ z = f(x, y) = \frac{\sin(\frac{\pi x}{4})}{\frac{\pi x}{4}} \cdot \frac{\sin(\frac{\pi y}{4})}{\frac{\pi y}{4}} \]

```matlab
» x = -10:.5:10; y = -10:.5:10;
» [x_grid,y_grid] = meshgrid(x,y);
» % MATLAB defines sin(pi*x)/(pi*x) as sinc(x)
» z = sinc(x_grid/4).*sinc(y_grid/4);
» surfl(x,y,abs(z)) % Surface plot with lighting effects
» title('Rectangular Diffraction Pattern Magnitude', ... 
  'fontsize',16)
» ylabel('y','fontsize',14)
» xlabel('x','fontsize',14)
```

Rectangular Diffraction Pattern Magnitude
Data Analysis Functions

The analysis of data coming from analytical calculations, computer simulations, or actual experimental data, is a requirement in most all engineering projects. With MATLAB we have a powerful set of predefined data analysis, and as we will see later, we can also write custom functions as needed.

Simple Analysis

- The first group of data analysis functions to consider finds maximums, minimums, sums, and products of vectors or columns of matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{max}(x) )</td>
<td>Returns ( y ) the largest value in vector ( x ) or each column of ( x ). Optionally ( k ) is the index where the max occurs.</td>
<td>( \text{max}(x, y) )</td>
<td>Returns a matrix full of the larger of the two values in ( x ) and ( y ) at the corresponding matrix indices.</td>
</tr>
<tr>
<td>([y, k] = \text{max}(x))</td>
<td></td>
<td>( \text{min}(x, y) )</td>
<td>Same as ( \text{max}(x, y) ) except minimum.</td>
</tr>
<tr>
<td>( \text{min}(x) )</td>
<td>Same as ( \text{max}(x) ) except minimum.</td>
<td>( \text{min}(x) )</td>
<td>Same as ( \text{max}(x, y) ) except minimum.</td>
</tr>
</tbody>
</table>
Table 3.6: Simple data analysis functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{sum}(x) )</td>
<td>The scalar (vector of column sums) ( \sum_{n=1}^{N} x(n) )</td>
<td>( \text{prod}(x) )</td>
<td>The scalar (vector of column products) ( \prod_{n=1}^{N} x(n) )</td>
</tr>
<tr>
<td>( \text{cumsum}(x) )</td>
<td>The running or cumulative sum version of the above, hence a vector or matrix</td>
<td>( \text{cumprod}(x) )</td>
<td>The running or cumulative product version of the above, hence a vector or matrix</td>
</tr>
</tbody>
</table>

**Example:**

```matlab
» x = 0:10;
» x_max = max(x); x_min = min(x);
» x_sum = sum(x); x_prod = prod(x)
» [x_max x_min x_sum x_prod]
ans =
    10     0    55     0
» cumsum(x)
ans =
   0     1     3     6    10    15    21    28    36
    45    55
```
```matlab
» cumprod(x)
ans =
     0     0     0     0     0     0     0     0     0     0
     0     0 % Why zeros?
```

**Sample Statistics**

- When the data we are operating on is viewed as the results of an experiment or the sampling of a *population*, then we may be interested in *sample statistics*

**Table 3.7: Sample statistics**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `mean(x)`         | The mean or average of the elements in `x`
|                   | \[
|                   | \mu = \frac{1}{N} \sum_{n=1}^{N} x(n) \]
|                   | \[
|                   | = \text{cumsum}(x) / \text{length}(x) \]
|                   | The column mean for `x` a matrix                  |
| `median(x)`       | The central or median value of `x`                |
| `sort(x)`         | Returns vector `x` with values arranged in ascending order; columns in ascending order for `x` a matrix |
| `var(x)`          | The average squared variation of `x` about its mean
|                   | \[
|                   | \sigma^2 = \frac{1}{N-1} \sum_{n=1}^{N} [x(n) - \mu]^2 \]
| `std(x)`          | The square root of the variance = \sigma         |
Examples: Practice! p. 93, 2 & 4

Determine the matrices represented by the following function references. The use MATLAB to check your answers. Assume that \( w \), \( x \), and \( y \) are the following matrices:

\[
w = \begin{bmatrix} 0 & 3 & -2 & 7 \end{bmatrix}, \quad x = \begin{bmatrix} 3 & -1 & 5 & 7 \end{bmatrix}
\]

\[
y = \begin{bmatrix} 1 & 3 & 7 \\ 2 & 8 & 4 \\ 6 & -1 & -2 \end{bmatrix}
\]

2. \( \text{min}(y) \)
   - Since \( y \) is a matrix the \text{min} function will operate on each column to produce a row vector whose elements are the minimum entry in each column of \( y \), hence we expect that

\[
\text{min}(y) = \begin{bmatrix} 1 & -1 & -2 \end{bmatrix}
\]

\[
\text{min}(y) = \begin{bmatrix} 1 & 3 & 7; & 2 & 8 & 4; & 6 & -1 & -2 \end{bmatrix};
\]

\[
\text{min}(y)
\]

\[
\text{ans} =
\]

\[
1 & -1 & -2
\]

4. \( \text{mean}(y) \)
   - Again since \( y \) is a matrix the \text{mean} function will operate on each column to produce a row vector whose elements are the average of each column of \( y \), hence we expect that
\[
\text{mean}(y) = \left[ \frac{(1 + 2 + 6)}{3} \; \frac{(3 + 8 - 1)}{3} \; \frac{(7 + 4 - 2)}{3} \right]
\]
\[
= \left[ 3.0 \; 3.33 \; 3.0 \right]
\]

\texttt{» mean(y)}
\[
\text{ans} =
\]
\[
3.0000 \; 3.3333 \; 3.0000
\]

**Histograms**

- A histogram is a plot of the distribution of the data, that is the number of data elements falling into *bins* formed by making contiguous intervals of the *x*-axis

- In figuring out grades for a course a histogram may be used to see how the students scores (input data) are distributed over a scale from 0 to 100; the bins in this case might be intervals of 10 points each

- MATLAB produces a histogram plot for a data set using the function \texttt{hist(x,n)} or \texttt{hist(x,bin\_centers)}
  - For the parameter \texttt{n} a scalar, it controls the number bins to be displayed
  - For the parameter \texttt{bin\_centers} a vector, it controls the location of the bin centers

**Example:** A histogram of a data set composed of random numbers chosen uniformly on the interval [0,1]
```matlab
» x = rand(1,10000); % generates unif. rand. #s on [0,1]
» hist(x,10)
» title('Histogram of 10,000 Uniform on [0,1] Random Numbers','fontsize',14)
» ylabel('Bin Count','fontsize',14')
» xlabel('Bin Value - x','fontsize',14')

Histogram of 10,000 Uniform on [0,1] Random Numbers

Ideally we should have 1000 in each bin

Flow Control using a Selection Statement

• In all programming languages there are means (usually more than one) to control the flow of the program

• The most common means of flow control is via an if statement/code block
The MATLAB if Statement

• The full form of the if code block is

```matlab
if logical_expression_#1
    statements
elseif logical_expression_#2
    statements
    .
    .
    .
elseif logical_expression_#N
    statements
else  % Default code to execute is in here
    statements
end
```

• The general if code block shown above is very powerful, but all the features that it offers may be confusing at first, and in fact are often not used.

• In flow chart form a simple flow control may be the following:

```
Logical Expression

---------
g < 50
---------

yes

no

count = count+1

disp(g)
```
• The MATLAB code is the following:

```matlab
%some program statements
if g < 50  % Note statements inside if block are
    % indented to improve readability.
    count = count + 1;
    disp(g);
end
%more program statements
```

• In an `if` code block decisions are made as a result of evaluating a logical expression(s), e.g.,

```matlab
if logical_expression
    run_this_code
end
```

– If the logical expression is true, then the code placed between the `if` and `end` statements is evaluated, otherwise we skip over it to the first code statement following the `end` statement

• Before studying more `if` code block examples, we will briefly study the use of relational and logical operators to construct logical expressions

**Relational and Logical Operators**

Using *relational operators* we can construct a single logical expression. If needed, several logical expressions can be combined into one logical expression using *logical operators*.

• MATLAB supports six relational operators as described in Table 3.8
Table 3.8: Relational operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose/Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Used to form the logical expression $a &lt; b$ which returns 1 if true and 0 if false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Used to form the logical expression $a \leq b$ which returns 1 if true and 0 if false</td>
</tr>
<tr>
<td>&gt;</td>
<td>Used to form the logical expression $a &gt; b$ which returns 1 if true and 0 if false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Used to form the logical expression $a \geq b$ which returns 1 if true and 0 if false</td>
</tr>
<tr>
<td>==</td>
<td>Used to form the logical expression $a = b$ which returns 1 if true and 0 if false</td>
</tr>
<tr>
<td>~=</td>
<td>Used to form the logical expression $a \neq b$ which returns 1 if true and 0 if false</td>
</tr>
</tbody>
</table>

- When the logical operators are used in expressions involving vectors or matrices, the returned value is a vector or matrix containing 0’s and 1’s respectively

**Example:** Let $a = \begin{bmatrix} 1 & 5 & 8 \end{bmatrix}$ and $b = \begin{bmatrix} 2 & 3 & 8 \end{bmatrix}$

```matlab
» a = [1 5 8]; b = [2 3 8];
» a < b
ans =
    1     0     0
» a > b
ans =
    0     1     0
```
```plaintext
» a == b
ans =
0 0 1
» a ~= b
ans =
1 1 0

- More complex logical expressions can be obtained by combining several sub expressions using the logical operators given in Table 3.9

Table 3.9: Logical operators

<table>
<thead>
<tr>
<th>Operator/Symbol</th>
<th>Purpose/Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>not, ~</td>
<td>not (a) or ~a returns inverts a 1 to 0 and a 0 to a 1</td>
</tr>
<tr>
<td>and, &amp;</td>
<td>and (a, b) or a&amp;b returns 1 if a and b are both true (1), otherwise 0</td>
</tr>
<tr>
<td>or,</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>xor (a, b) returns 1 if a and b are different, otherwise 0</td>
</tr>
</tbody>
</table>

Example: Continuation of previous example

» a = [1 5 8]; b = [2 3 8];
» (a < b)
ans =
1 0 0
» ~(a < b)
ans =
0 1 1
```
Example: Practice! p. 99 (1,5,7)

Determine if the following expressions are true or false. Then check your answers using MATLAB.

\[ a = 5.5, \ b = 1.5, \ k = -3 \]

» \( a = 5.5; \ b = 1.5; \ k = -3; \)

1. \( a < 10.0 \)

By inspection the statement \( a < 10.0 \) is true, or logical 1

» \( a < 10.0 \)

ans = 1

5. \( \neg (a == 3 \times b) \)

We know that \( a = 5.5 \) and \( 3b = 4.5 \), thus the interior logical expression is false (logical 0), and its complement is true (logical 1)

» \( \neg (a == 3 \times b) \)

ans = 1

7. \( a < 10 \quad \& \quad a > 5 \)

By inspection \( a < 10 \) is true and \( a > 5 \) is true, the logical and of these to expressions is true or logical 1

» \( a < 10 \quad \& \quad a > 5 \)

ans = 1
More if Code Blocks

In this section we study the else and elseif clauses that are part of the complete if code block, and also consider nested if code blocks

- The stripped down if code block discussed earlier is useful when an operation is either performed or not performed
- When say two or more different operations are desired, depending upon a corresponding set of logical expression outcomes, we can use the full if code block

Example: Suppose we want to conditionally evaluate

\[ y = \begin{cases} 
2x, & x > 4 \\
0, & x = 4 \\
-5x, & \text{otherwise} 
\end{cases} \quad (3.1) \]

- In flow chart form we have

---

Chapter 3: Data Analysis Functions
• In MATLAB we have:
  
  ```matlab
  if x > 4
    y = 2*x;
  elseif x == 4
    y = 0;
  else
    y = -5*x;
  end
  ```

**The switch (case) Code Block**

• New to MATLAB 5 is the switch or case code block

• Most all high level languages provide the program flow construct, so it only seems reasonable that MATLAB has adopted it as well

• The basic form of the switch statement is

  ```matlab
  switch switch_expr
    case case_expr,
      statement, ..., statement
    case {case_expr1, case_expr2, case_expr3,...}
      statement, ..., statement
    ...
    otherwise,
      statement, ..., statement
  end
  ```

  – The basic functionality of the switch statement is to
  `switch` to the first code following a particular case statement where the switch expression and case expressions match
– The code following the `otherwise` statement is run only if no match is found in the above case statements.
– Code execution continues following the `end` statement.

**Example:** Adjust program flow based on a user entered text string.

% MATLAB script file switch_demo.m
% Demo of the Switch (Case) Code Block
input_text = ...
    input('Enter some text in single quotes: '); %Make sure case is not an issue:
input_text = lower(input_text);
switch input_text
    case 'yes'
        disp('You have answered the question with a yes.');
    case 'no'
        disp('You have answered the question with a no.');
    otherwise
        disp('You have not answered the question with a yes or no.');
end

• Sample output

```matlab
» switch_demo
Enter Some Text in single quotes: 'help'
You have not answered the question with a yes or no.
» switch_demo
Enter Some Text in single quotes: 'NO'
You have answered the question with a no.
```
Example: Practice! p. 101 (2,4)

Provide a MATLAB code block that performs the steps indicated.

2. If \( \log(x) > 3 \) set \( \text{time} \) equal to 0 and increment \( \text{count} \) by one
   
   ```
   if log(x) > 3
       time = 0;
       count = count + 1;
   end
   ```

4. If \( \text{dist} \geq 100.0 \) increment \( \text{time} \) by 2.0. If \( 50 < \text{dist} < 100 \) increment \( \text{time} \) by 1. Otherwise increment \( \text{time} \) by 0.5.
   
   ```
   if dist >= 100.0
       time = time + 2.0;
   elseif (dist > 50) & (dist < 100)
       time = time + 1;
   else
       time = time + 0.5;
   end
   ```

Logical Functions

- Logical functions are useful in vectorizing algorithms for efficient execution in the MATLAB environment

- Six functions of particular interest are given in Table 3.10

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any(x)</td>
<td>Returns 1 if <strong>any</strong> element in x is nonzero, 0 otherwise. For a matrix returns a row vector whose elements are 1 if the corresponding column of x contains <strong>any</strong> nonzero element.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>all(x)</td>
<td>Returns 1 if all elements in x is nonzero, zero otherwise. For a matrix returns a row vector whose elements are 1 if the corresponding column of x contains all nonzero elements.</td>
</tr>
<tr>
<td>find(x)</td>
<td>Returns a vector of the indices of x corresponding to nonzero elements. For x a matrix x is first reshaped into a single column vector formed from the columns of the original x, then processed as before.</td>
</tr>
<tr>
<td>isnan(x)</td>
<td>Returns a matrix filled with 1’s where the elements of x are NaN and 0 otherwise.</td>
</tr>
<tr>
<td>finite(x)</td>
<td>Returns a matrix filled with 1’s where the elements of x are finite and 0 if they are infinite or NaN.</td>
</tr>
<tr>
<td>isempty(x)</td>
<td>Returns 1 if x is an empty matrix, 0 otherwise.</td>
</tr>
</tbody>
</table>
• The `find` function is particularly useful in vectorizing the evaluation of expressions such as (3.1)

• We can modify the code block on p. 39 and place it into an m-file as follows:

```matlab
% MATLAB script file eqn_3_1.m
% Vectorize equation (3-1)
% Work x > 4 case:
indices = find(x > 4);
y(indices) = 2*x(indices);
% Work x == 4 case:
indices = find(x == 4);
y(indices) = 0*x(indices);
% Work x < 4 case:
indices = find(x < 4);
y(indices) = -5*x(indices);
```

• Usage of `eqn_3_1.m`:

```matlab
» x = -20:.1:20;
» eqn_3_1
» plot(x,y)
» eqn_3_1
» plot(x,y)
» plot(x,x,x,y)
» grid
» title('Plot of Equation (3-1) using find()','fontsize',16)
» ylabel('y and x','fontsize',14)
» xlabel('x','fontsize',14)
```
**Example:** Practice! p. 103 (4,6)

Determine the value of the following expressions. Then check your answers by entering the expressions into MATLAB. Assume that

\[ y = \begin{cases} 
2x, & x > 4 \\
0, & x = 4 \\
-5x, & \text{otherwise} 
\end{cases} \]

4. `any(all(b))`

First off the `all()` function returns a three element row vec-
tor containing 1’s only if each column contains nonzero entries, then \texttt{any()} returns a scalar 1 if any entry in the row vector is nonzero. By inspection \( \texttt{all(b)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \). Finally

\[
\text{any}(\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}) = 0
\]

\[
\texttt{any(all(b))}
\]

\[
\texttt{ans} = 0
\]

\[
\texttt{all(b) \% Just checking}
\]

\[
\texttt{ans} = 0 \quad 0 \quad 0
\]

6. \texttt{any(b(1:2,1:3))}

The matrix that \texttt{any} is operating on is

\[
c = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 0 & 3 \end{bmatrix}
\]

When \texttt{any} operates on \( c \) we obtain a row vector containing 1 only if a nonzero entry is found in one of the columns, thus

\[
\text{any}(b(1:2,1:3)) = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}
\]

\[
\texttt{any(b(1:2,1:3))}
\]

\[
\texttt{ans} = 1 \quad 0 \quad 1
\]