The McGraw-Hill Companies

PowerPoint to accompany

Introduction to MATLAB for Engineers, Third Edition

William J. Palm III

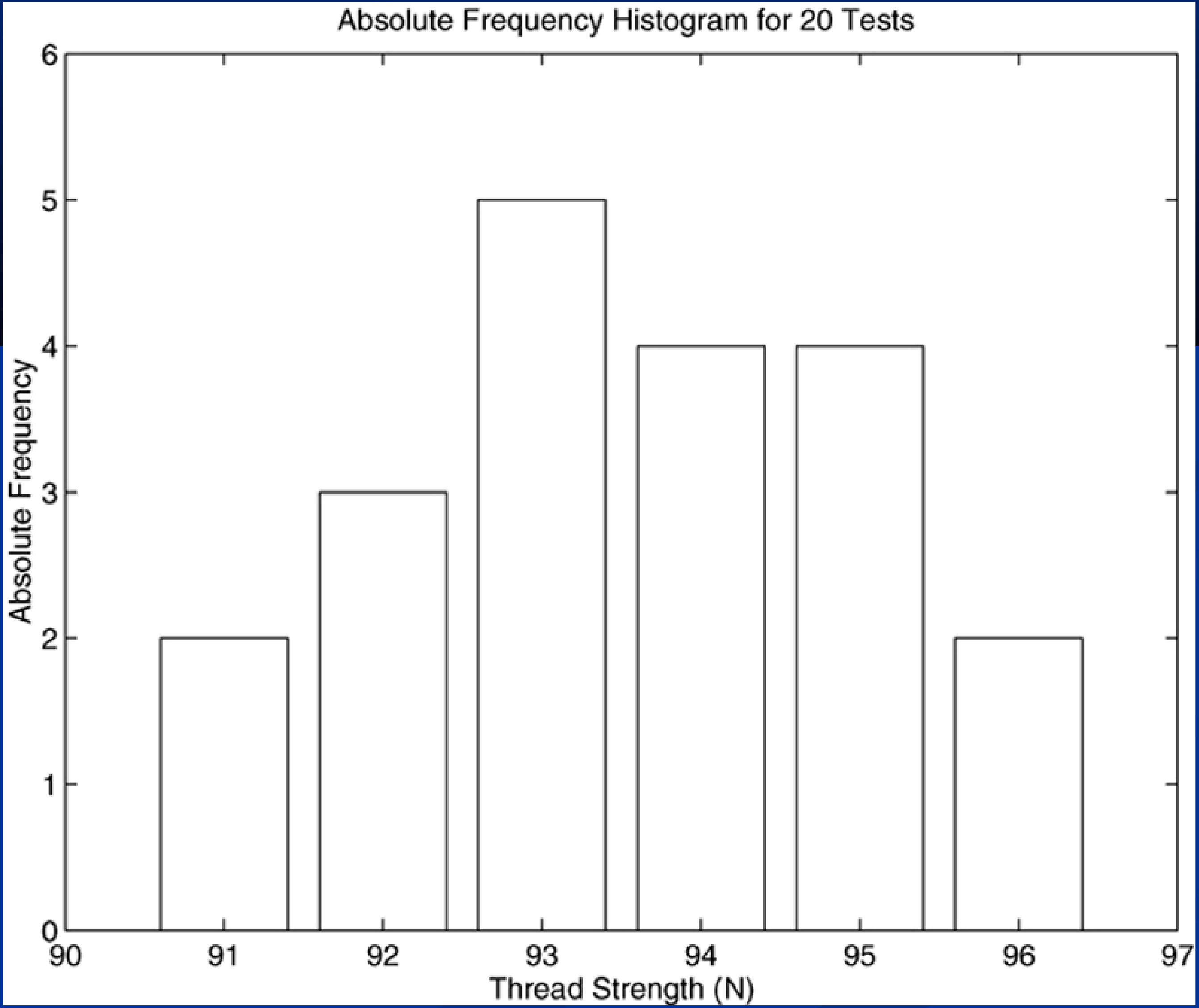
Chapter 7
Statistics, Probability, and Interpolation

Breaking Strength of Thread

```
% Thread breaking strength data for 20 tests.
y = [92,94,93,96,93,94,95,96,91,93,...
95,95,95,92,93,94,91,94,92,93];
% The six possible outcomes are ...
91,92,93,94,95,96.
x = [91:96];
hist(y,x),axis([90 97 0 6]),...
ylabel('Absolute Frequency'),...
xlabel('Thread Strength (N)'),...
title('Absolute Frequency Histogram...
for 20 Tests')
```

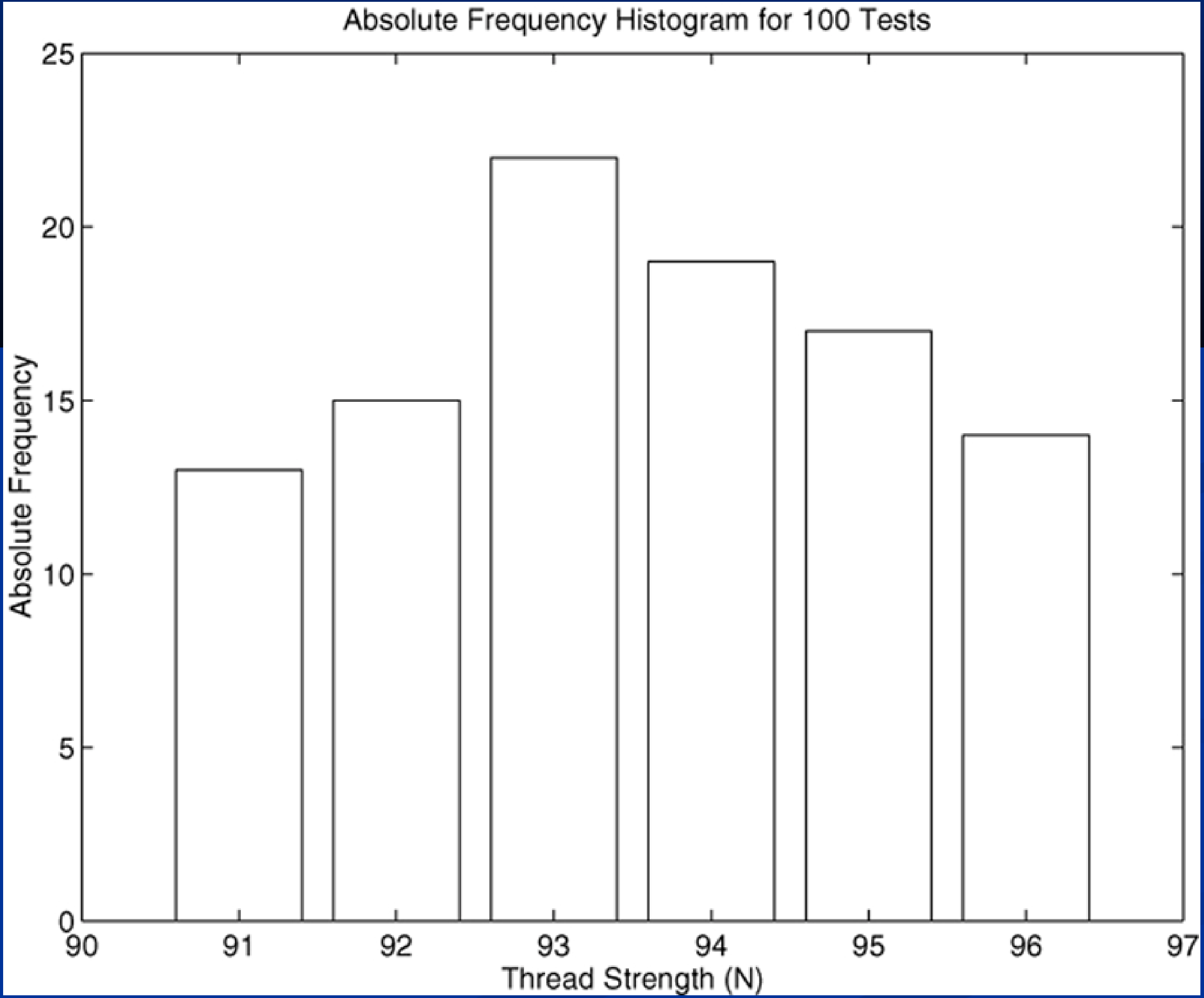
This creates the next figure.

Histograms for 20 tests of thread strength. Figure 7.1–1, page 297



Absolute frequency histogram for 100 thread tests.

Figure 7.1–2. This was created by the program on page 298.



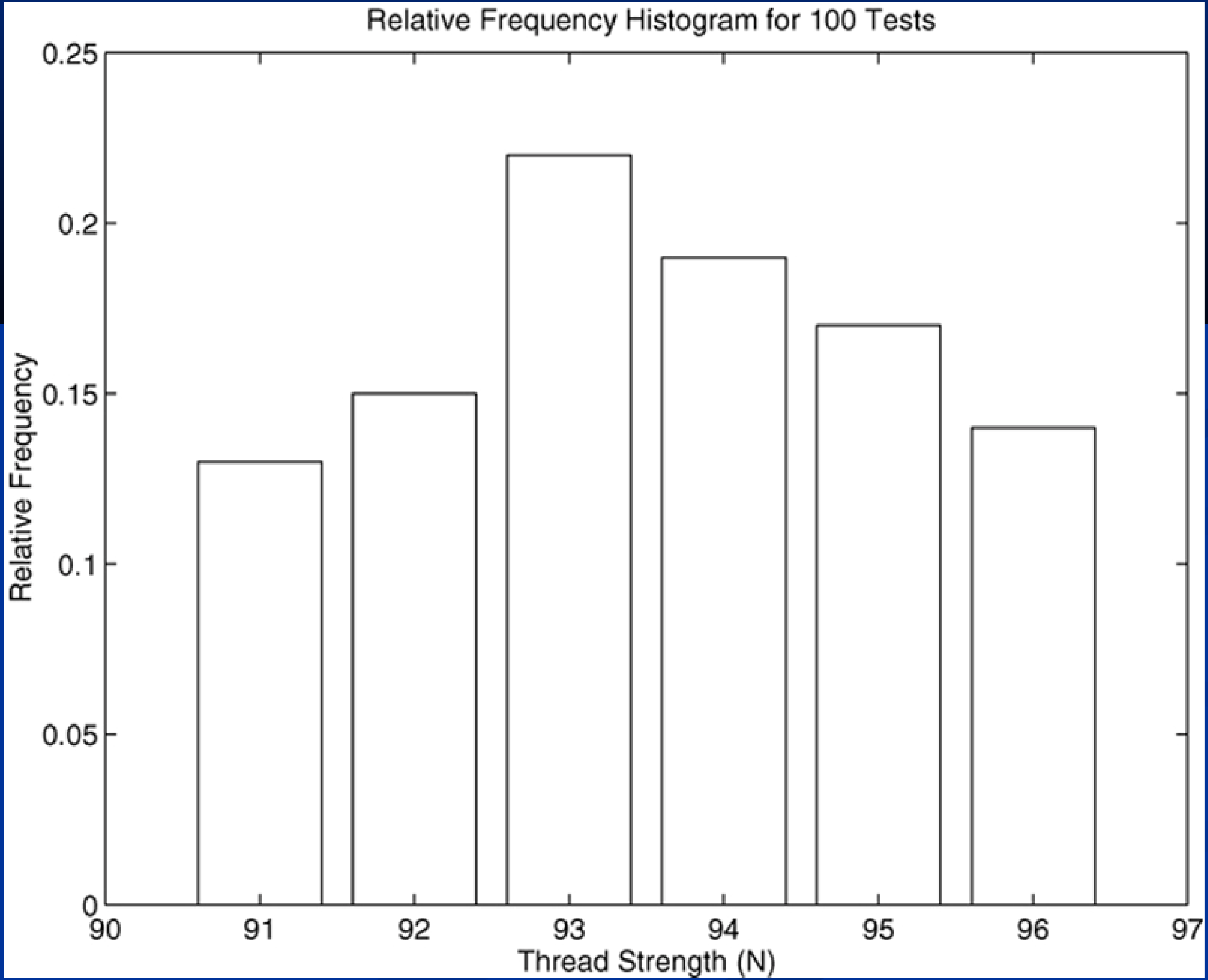
Use of the `bar` function for relative frequency histograms (page 299).

```
% Relative frequency histogram using ...
the  bar function.
tests = 100;
y = [13,15,22,19,17,14]/tests;
x = [91:96];
bar(x,y),ylabel('Relative Frequency'),...
xlabel('Thread Strength (N)'),...
title('Relative Frequency Histogram ...for
100 Tests')
```

This creates the next figure.

Relative frequency histogram for 100 thread tests.

Figure 7.1–3



Use of the `hist` function for relative frequency histograms.

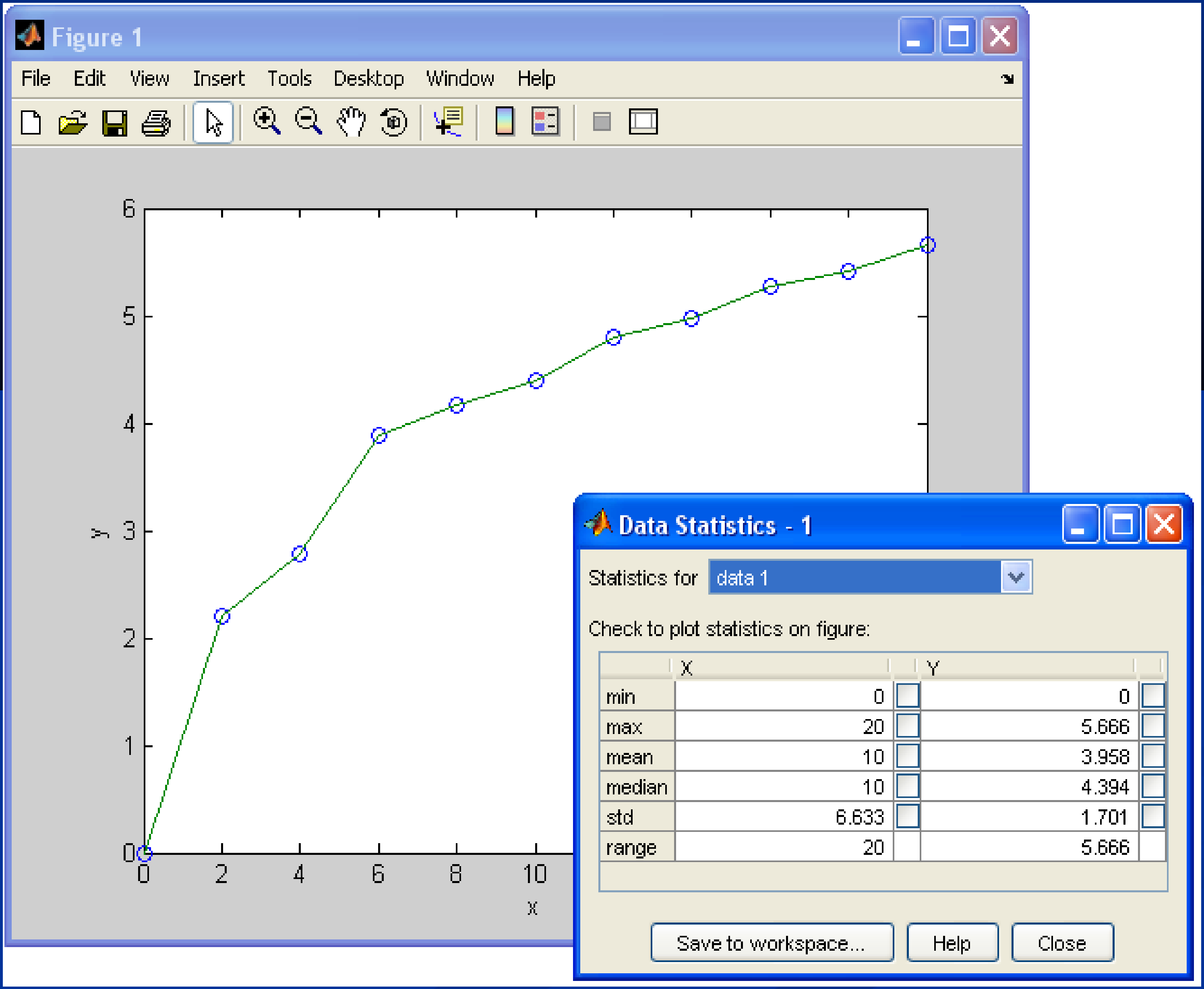
```
tests = 100;
y =
[91*ones(1,13),92*ones(1,15),93*ones(1,22),...
94*ones(1,19),95*ones(1,17),96*ones(1,14)];
x = [91:96];
[z,x] = hist(y,x);bar(x,z/tests),...
ylabel('Relative Frequency'),xlabel('Thread
Strength (N)'),...
title('Relative Frequency Histogram for 100
Tests')
```

This also creates the previous figure.

Histogram functions Table 7.1–1

Command	Description
`bar(x,y)`	Creates a bar chart of y versus x.
`hist(y)`	Aggregates the data in the vector y into 10 bins evenly spaced between the minimum and maximum values in y.
`hist(y,n)`	Aggregates the data in the vector y into n bins evenly spaced between the minimum and maximum values in y.
`hist(y,x)`	Aggregates the data in the vector y into bins whose center locations are specified by the vector x. The bin widths are the distances between the centers.
`[z,x] = hist(y)`	Same as `hist(y)` but returns two vectors z and x that contain the frequency count and the bin locations.
`[z,x] = hist(y,n)`	Same as `hist(y,n)` but returns two vectors z and x that contain the frequency count and the bin locations.
`[z,x] = hist(y,x)`	Same as `hist(y,x)` but returns two vectors z and x that contain the frequency count and the bin locations. The returned vector x is the same as the user-supplied vector x.

The Data Statistics tool. Figure 7.1–4 on page 301



Check to plot statistics on figure:

	X	Y
min	0	0
max	20	5.666
mean	10	3.958
median	10	4.394
std	6.633	1.701
range	20	5.666

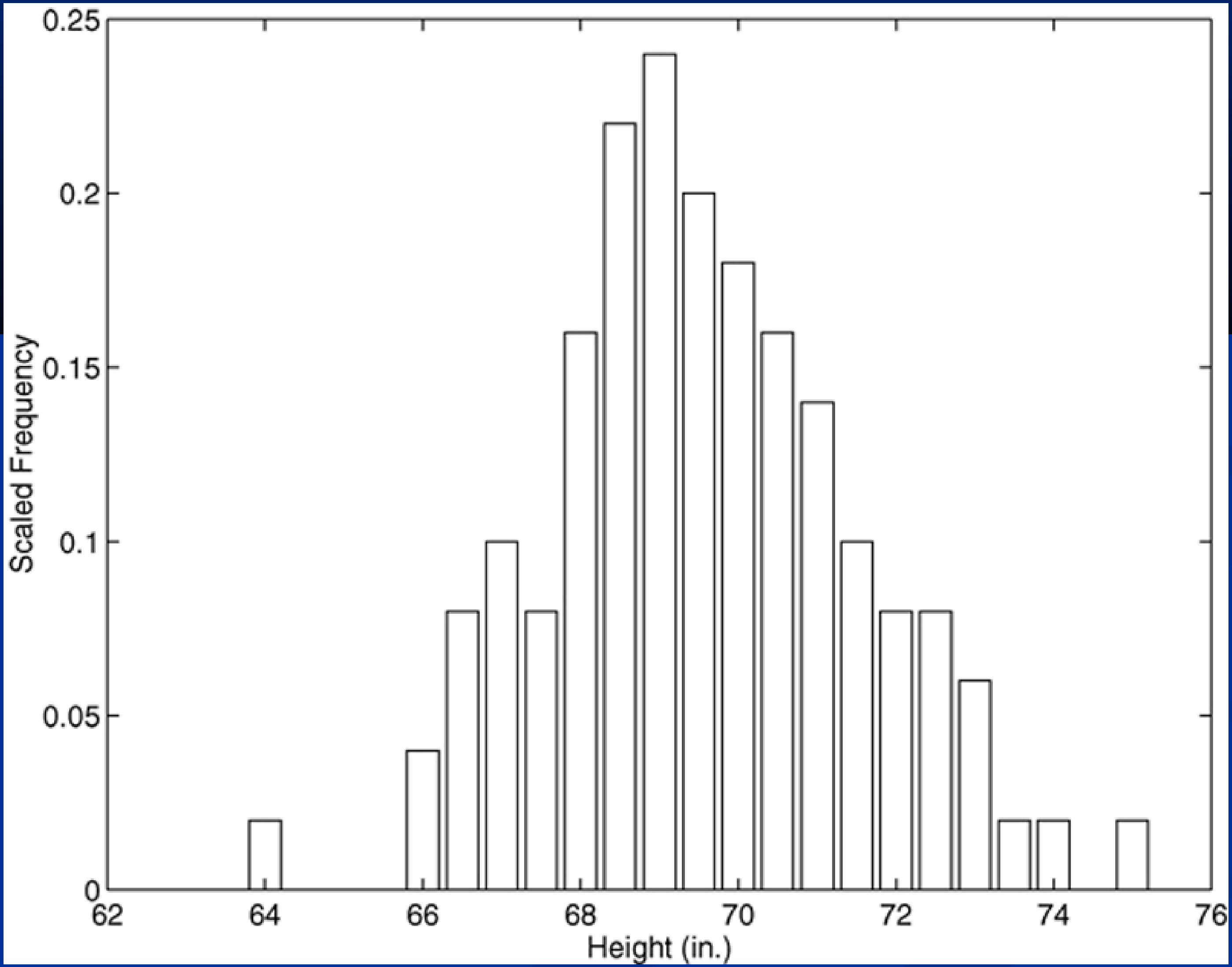
More? See page 300.

Scaled Frequency Histogram (pages 301-303)

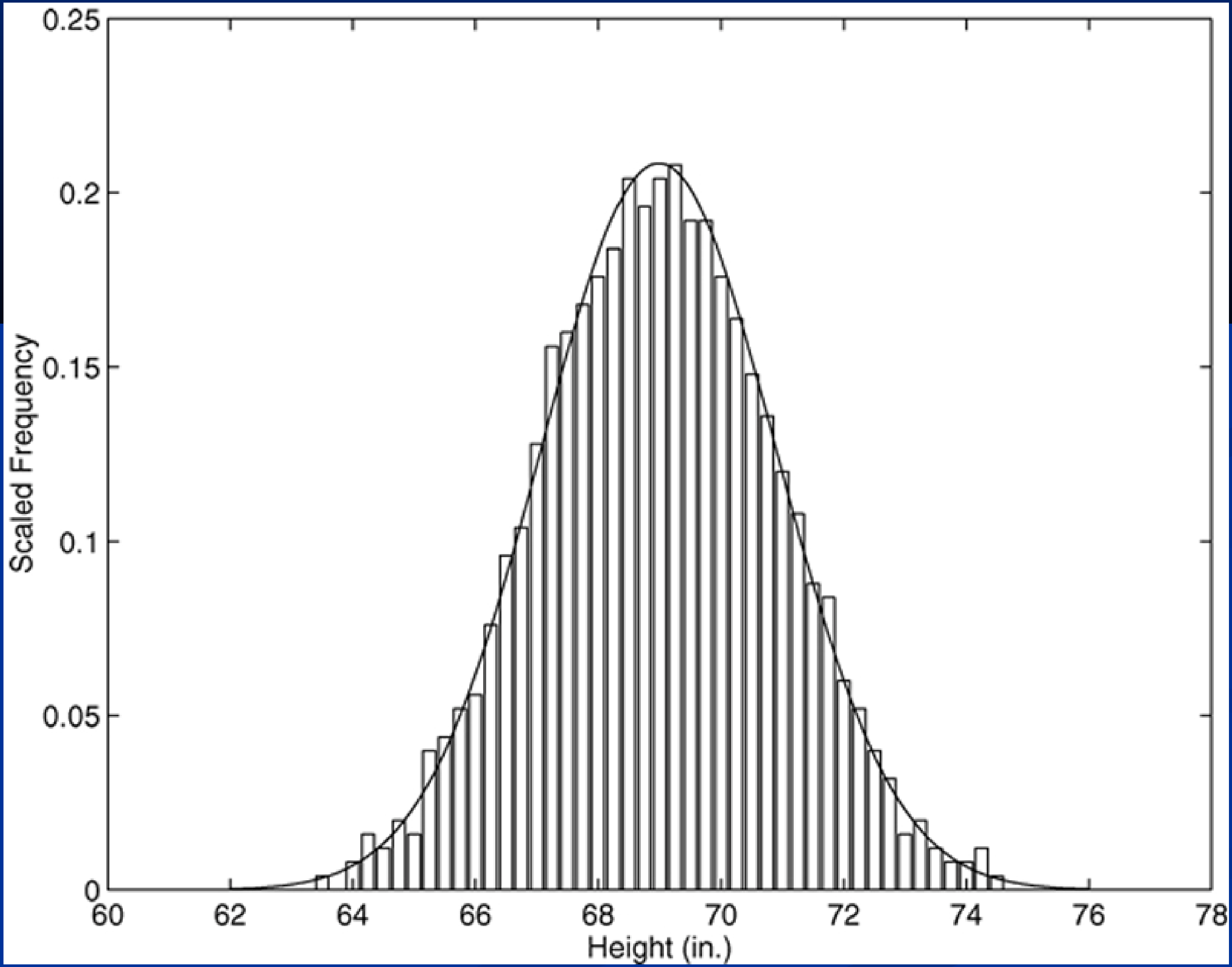
```
% Absolute frequency data.
y_abs=[1,0,0,0,2,4,5,4,8,11,12,10,9,8,7,5,4,4,3,1,1,0,1];
binwidth = 0.5;
% Compute scaled frequency data.
area = binwidth*sum(y_abs);
y_scaled = y_abs/area;
% Define the bins.
bins = [64:binwidth:75];
% Plot the scaled histogram.
bar(bins,y_scaled),...
ylabel('Scaled Frequency'),xlabel('Height (in.)')
```

This creates the next figure.

Scaled histogram of height data. Figure 7.2–1, page 302

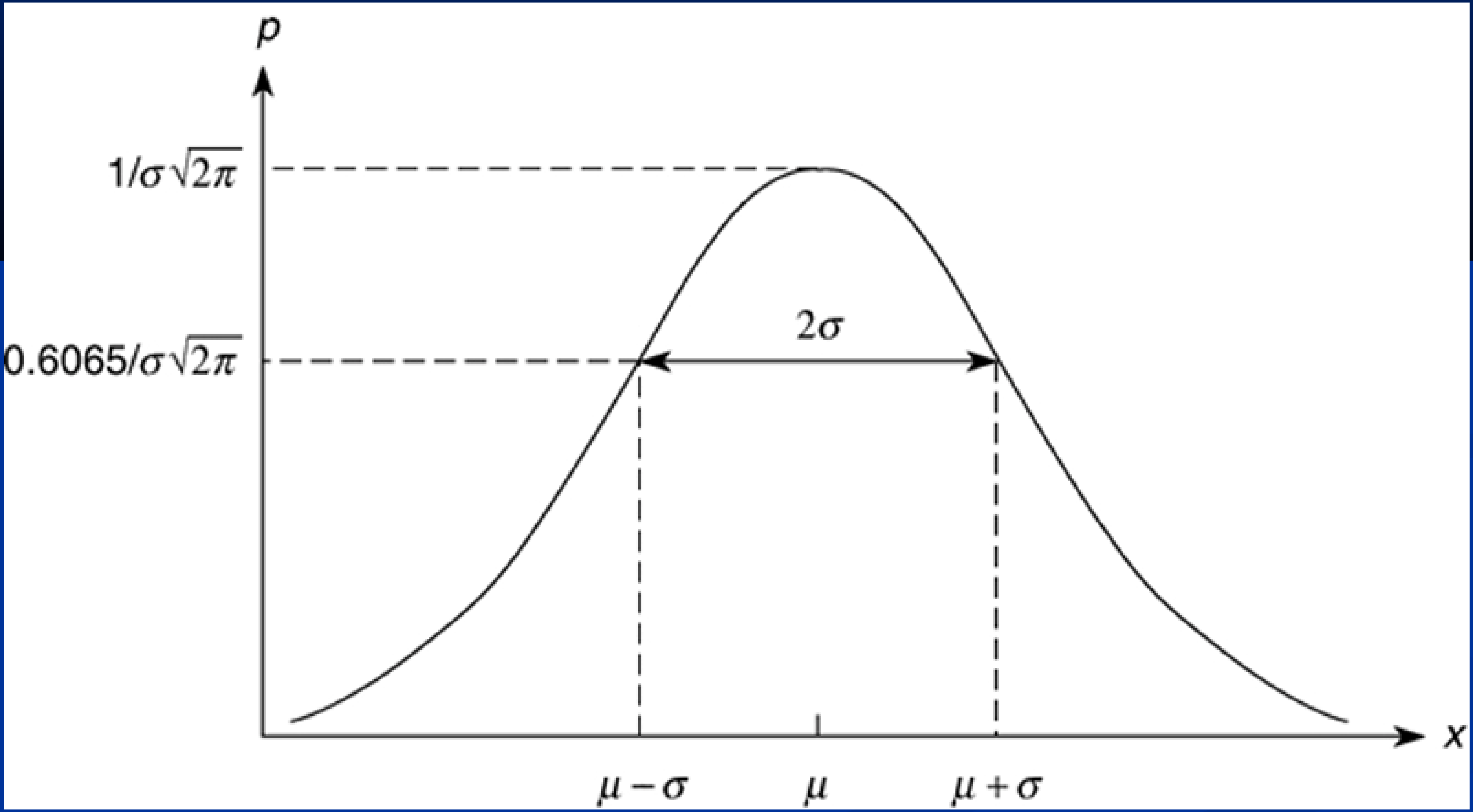


Scaled histogram of height data for very many measurements.



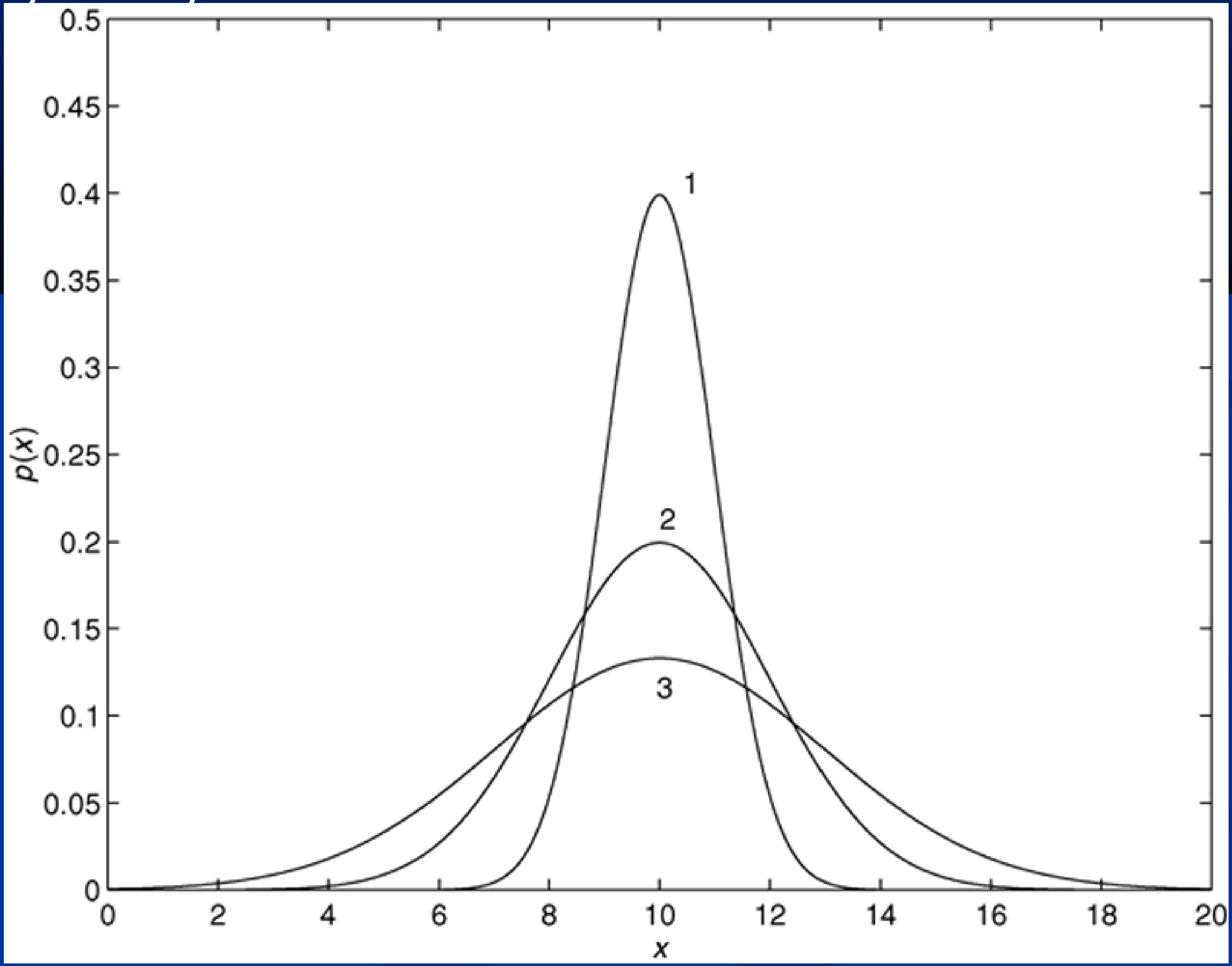
The basic shape of the normal distribution curve.

Figure 7.2–2, page 304

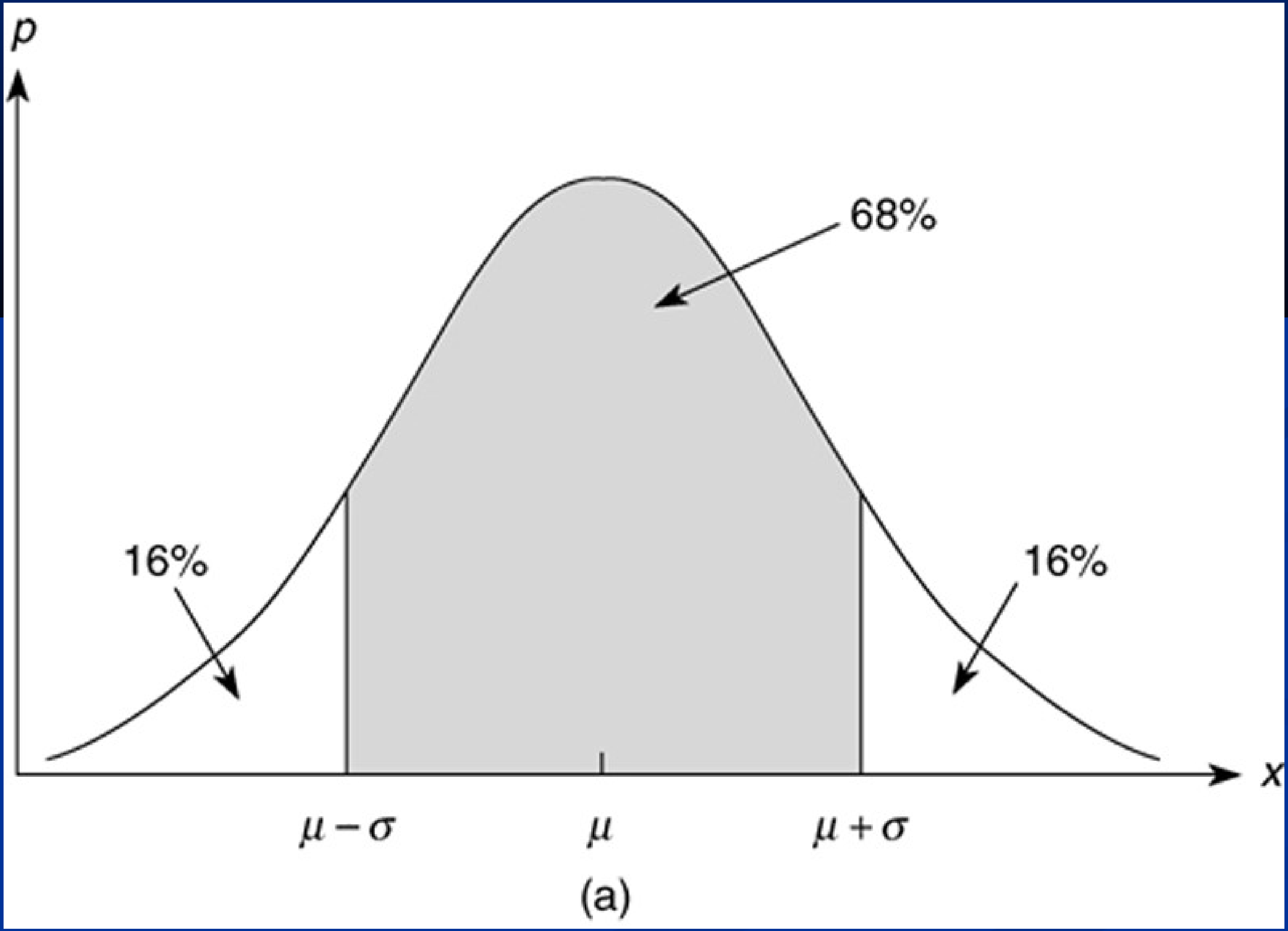


More? See pages 303-304.

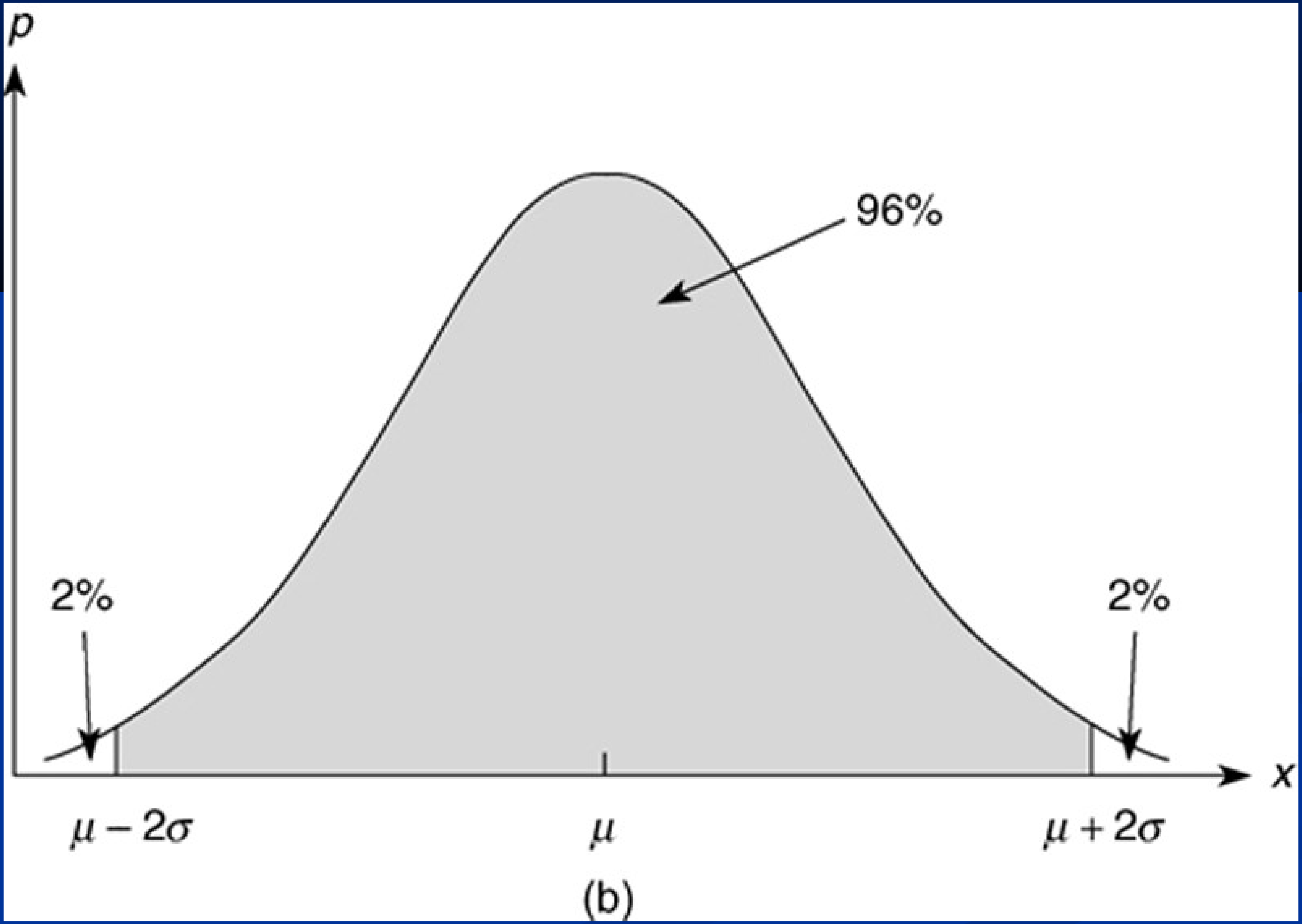
The effect on the normal distribution curve of increasing σ. For this case *μ* = 10, and the three curves correspond to σ = 1, *σ* = 2, and *σ* = 3.



Probability interpretation of the μ ± σ limits.



Probability interpretation of the μ ± 2σ limits.



More? See pages 431-432.

The probability that the random variable *x* is no less than *a* and no greater than *b* is written as $P(a \leq x \leq b)$. It can be computed as follows:

$$P(a \leq x \leq b) = \frac{1}{2}\left[\operatorname{erf}\left(\frac{b-\mu}{\sigma\sqrt{2}}\right) - \operatorname{erf}\left(\frac{a-\mu}{\sigma\sqrt{2}}\right)\right] \qquad (7.2\text{–}3)$$

See pages 305-306.

Sums and Differences of Random Variables (page 307)

It can be proved that the mean of the sum (or difference) of two independent normally distributed random variables equals the sum (or difference) of their means, but the variance is always the sum of the two variances. That is, if x and y are normally distributed with means μ_x and μ_y, and variances σ_x^2 and σ_y^2, and if $u = x + y$ and $\upsilon = x - y$, then

$$\mu_u = \mu_x + \mu_y \qquad (7.2\text{–}4)$$

$$\mu_\upsilon = \mu_x - \mu_y \qquad (7.2\text{–}5)$$

$$\sigma_u^2 = \sigma_\upsilon^2 = \sigma_x^2 + \sigma_y^2 \qquad (7.2\text{–}6)$$

Random number functions Table 7.3–1

Command	**Description**
`rand`	Generates a single uniformly distributed random number between 0 and 1.
`rand(n)`	Generates an $n \times n$ matrix containing uniformly distributed random numbers between 0 and 1.
`rand(m,n)`	Generates an $m \times n$ matrix containing uniformly distributed random numbers between 0 and 1.
`s = rand('twister')`	Returns a 35-element vector `s` containing the current state of the uniformly distributed generator.
`rand('twister',s)`	Sets the state of the uniformly distributed generator to `s`.
`rand('twister',0)`	Resets the uniformly distributed generator to its initial state.
`rand('twister',j)`	Resets the uniformly distributed generator to state `j`, for integer `j`.
`rand('twister',sum(100*clock))`	Resets the uniformly distributed generator to a different state each time it is executed.

Table 7.3–1 (continued)

`randn`	Generates a single normally distributed random number having a mean of 0 and a standard deviation of 1.
`randn(n)`	Generates an $n \times n$ matrix containing normally distributed random numbers having a mean of 0 and a standard deviation of 1.
`randn(m,n)`	Generates an $m \times n$ matrix containing normally distributed random numbers having a mean of 0 and a standard deviation of 1.
`s = randn('state')`	Like `rand('twister')` but for the normally distributed generator.
`randn('state',s)`	Like `rand('twister',s)` but for the normally distributed generator.
`randn('state',0)`	Like `rand('twister',0)` but for the normally distributed generator.
`randn('state',j)`	Like `rand('twister',j)` but for the normally distributed generator.
`randn('state',sum(100*clock))`	Like `rand('twister',sum(100*clock))` but for the normally distributed generator.
`randperm(n)`	Generates a random permutation of the integers from 1 to n.

The following session shows how to obtain the same sequence every time `rand` is called.

```
>>rand(’twister’,0)
>>rand
ans =
     0.5488
>>rand
ans =
     0.7152
>>rand(’twister’,0)
>>rand
ans =
     0.5488
>>rand
ans =
     0.7152
```

You need not start with the initial state in order to generate the same sequence. To show this, continue the above session as follows.

```
>>s = rand('twister');
>>rand('twister',s)
>>rand
ans =
     0.6028
>>rand('twister',s)
>>rand
ans =
     0.6028
```

The general formula for generating a uniformly distributed random number y in the interval $[a, b]$ is

$$y = (b - a)\,x + a \qquad (7.3\text{–}1)$$

where x is a random number uniformly distributed in the interval [0, 1]. For example, to generate a vector `y` containing 1000 uniformly distributed random numbers in the interval [2, 10], you type `y = 8*rand(1,1000) + 2`.

If x is a random number with a mean of 0 and a standard deviation of 1, use the following equation to generate a new random number y having a standard deviation of σ and a mean of μ.

$$y = \sigma x + \mu \qquad (7.3\text{–}2)$$

For example, to generate a vector y containing 2000 random numbers normally distributed with a mean of 5 and a standard deviation of 3, you type

```
y = 3*randn(1,2000) + 5.
```

If y and x are linearly related, as

$$y = bx + c \qquad (7.3\text{–}3)$$

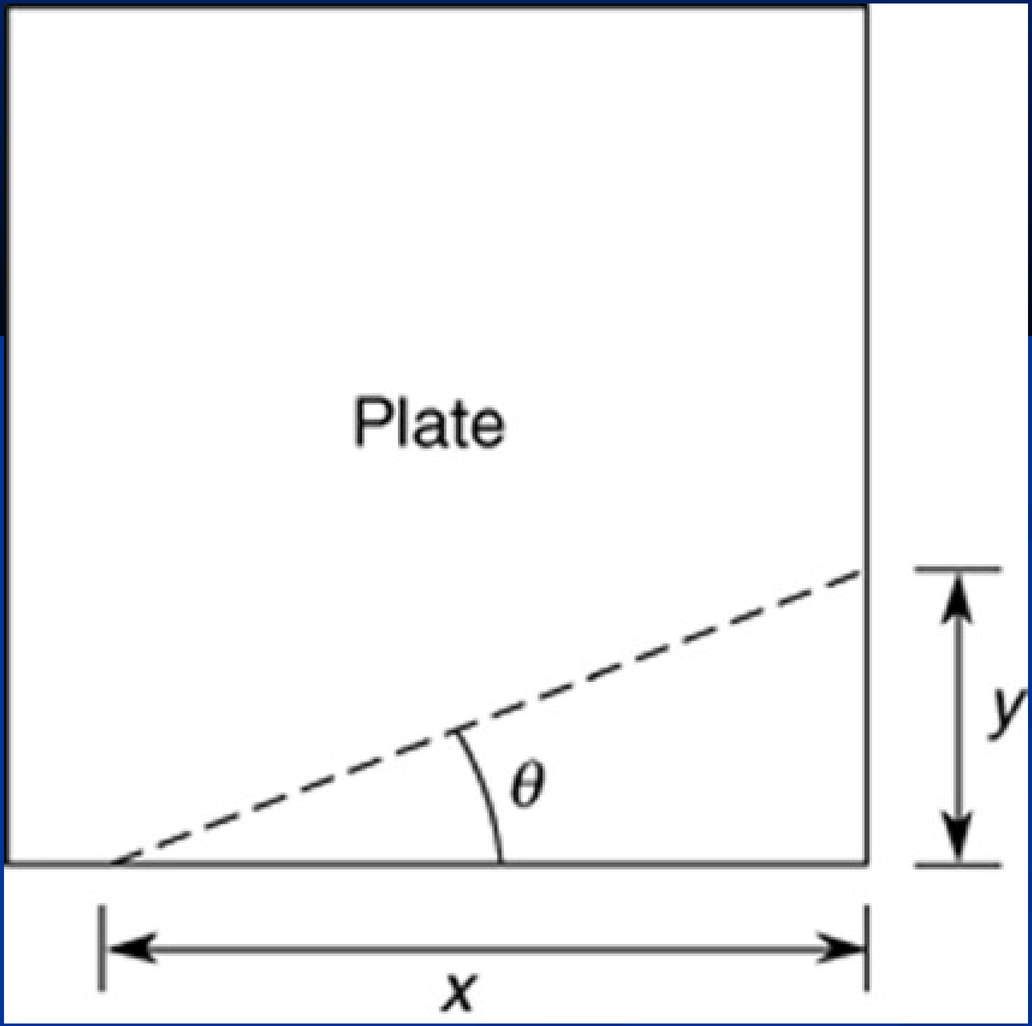
and if x is normally distributed with a mean μ_x and standard deviation σ_x, it can be shown that the mean and standard deviation of y are given by

$$\mu_y = b\mu_x + c \qquad (7.3\text{–}4)$$

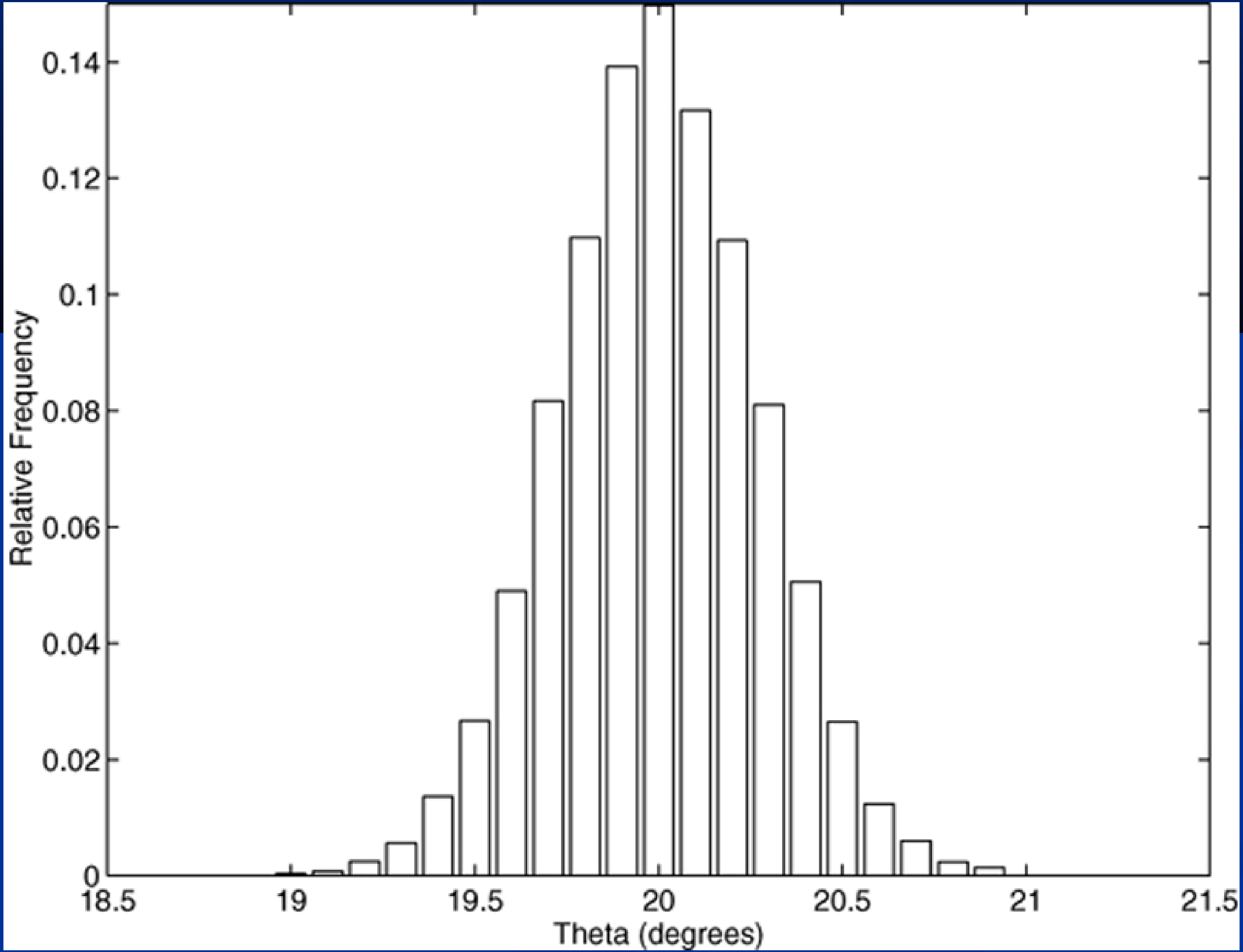
$$\sigma_y = |b|\,\sigma_x \qquad (7.3\text{–}5)$$

More? See pages 310-311.

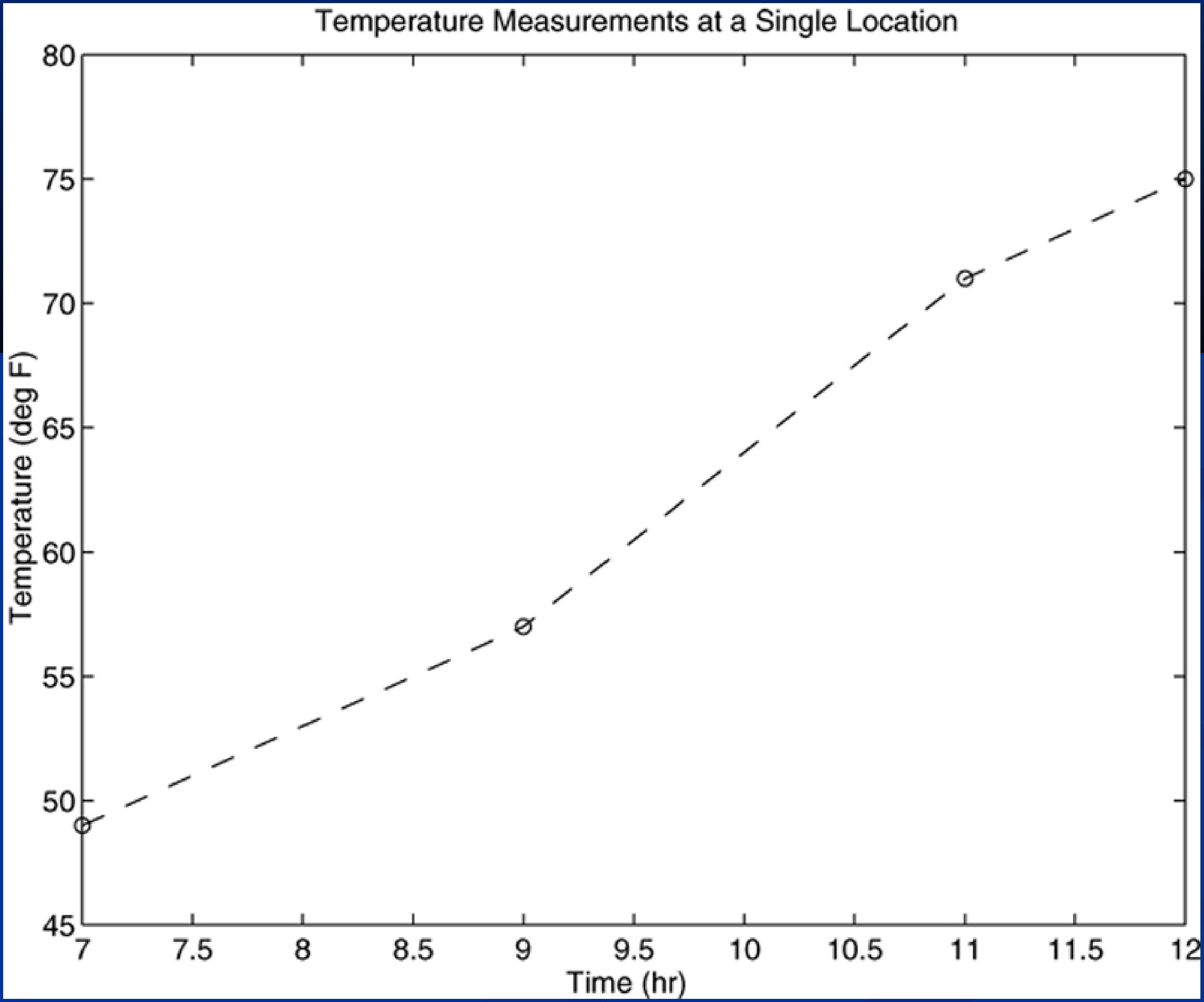
Statistical analysis and manufacturing tolerances: Example 7.3-1. Dimensions of a triangular cut. Figure 7.3–1 , page 312



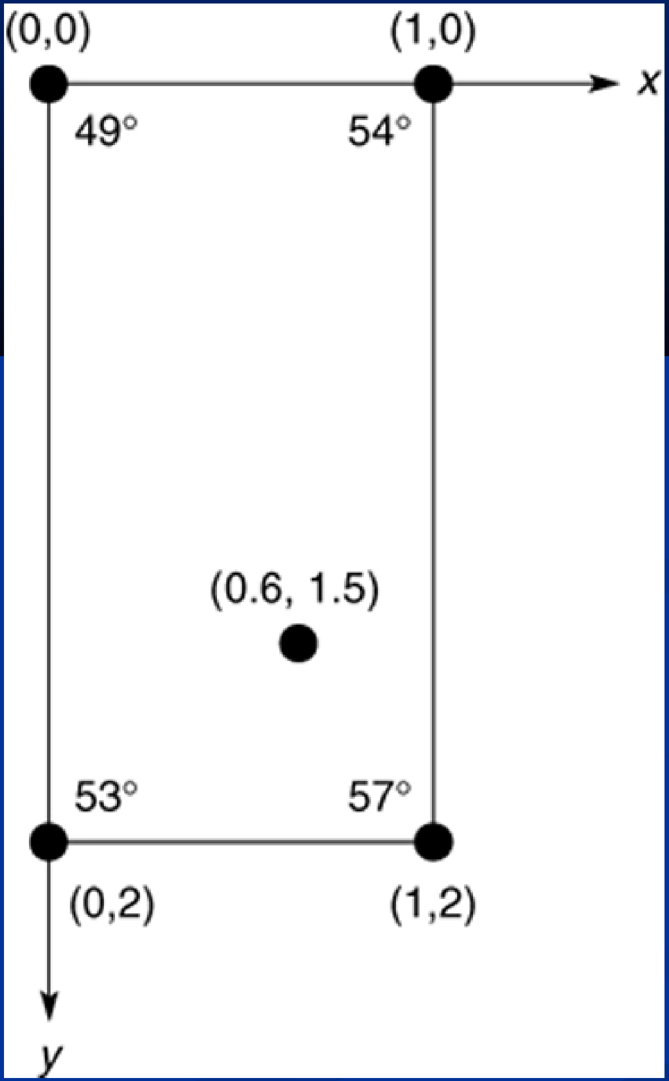
Scaled histogram of the angle θ. Figure 7.3–2, page 313



Applications of interpolation: A plot of temperature data versus time. Figure 7.4–1, page 314



Temperature measurements at four locations. Figure 7.4–2, page 316



More? See pages 313-317.

Linear interpolation functions. Table 7.4–1, page 317

Command	**Description**
`Y_int = interp1(x,y,x_int)`	Used to linearly interpolate a function of one variable: $y = f(x)$. Returns a linearly interpolated vector `y_int` at the specified value `x_int`, using data stored in `x` and `y`.

Table 7.4–1 Continued

```
Z_int = interp2(x,y,z,x_int,y_int)
```

Used to linearly interpolate a function of two variables: $y = f(x, y)$. Returns a linearly interpolated vector `z_int` at the specified values `x_int` and `y_int`, using data stored in `x`, `y`, and `z`.

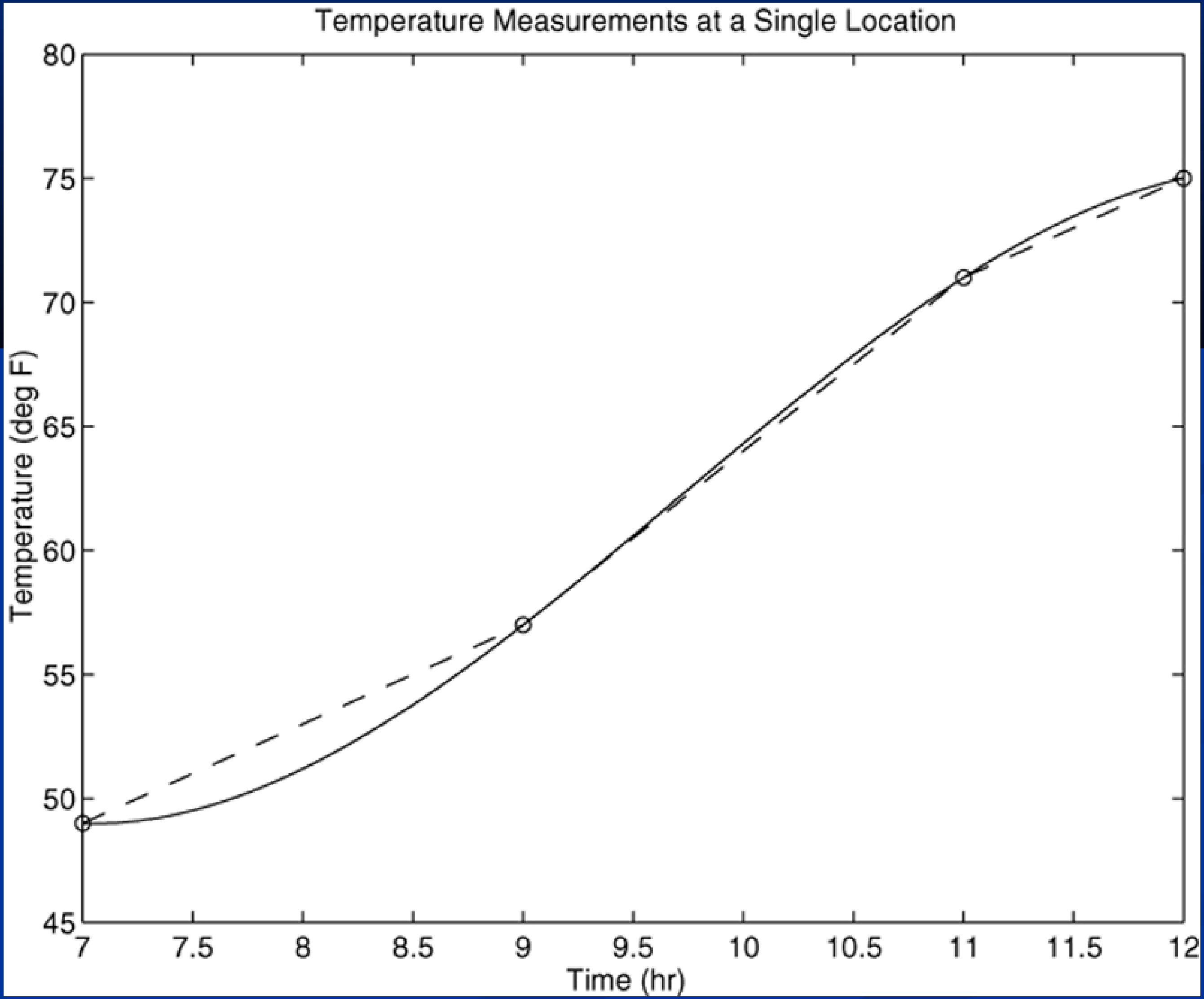
Cubic-spline interpolation: The following session produces and plots a cubic-spline fit, using an increment of 0.01 in the *x* values (pages 317-319).

```
>>x = [7,9,11,12];
>>y = [49,57,71,75];
>>x_int = 7:0.01:12;
>>y_int = spline(x,y,x_int);
>>plot(x,y,’o’,x,y,’--’,x_int,y_int),...
 xlabel(’Time (hr)’),ylabel(’Temperature
(deg F)’,...
 title(’Temperature Measurements at a
Single Location’),...
 axis([7 12 45 80])
```

This produces the next figure.

Linear and cubic-spline interpolation of temperature data.

Figure 7.4–3, page 319



Polynomial interpolation functions. Table 7.4–2, page 320

Command

```
y_est = interp1(x,y,x_est,'spline')
```

Description

Returns a column vector `y_est` that contains the estimated values of *y* that correspond to the *x* values specified in the vector `x_est`, using cubic-spline interpolation.

Table 7.4–2 Continued

```
y_int = spline(x,y,x_int)
```

Computes a cubic-spline interpolation where `x` and `y` are vectors containing the data and `x_int` is a vector containing the values of the independent variable *x* at which we wish to estimate the dependent variable *y*. The result `Y_int` is a vector the same size as `x_int` containing the interpolated values of *y* that correspond to `x_int`.

Table 7.4–2 Continued

```
y_int = pchip(x,y,x_int)
```

Similar to spline but uses piecewise cubic Hermite polynomials for interpolation to preserve shape and respect monotonicity.

Table 7.4–2 Continued

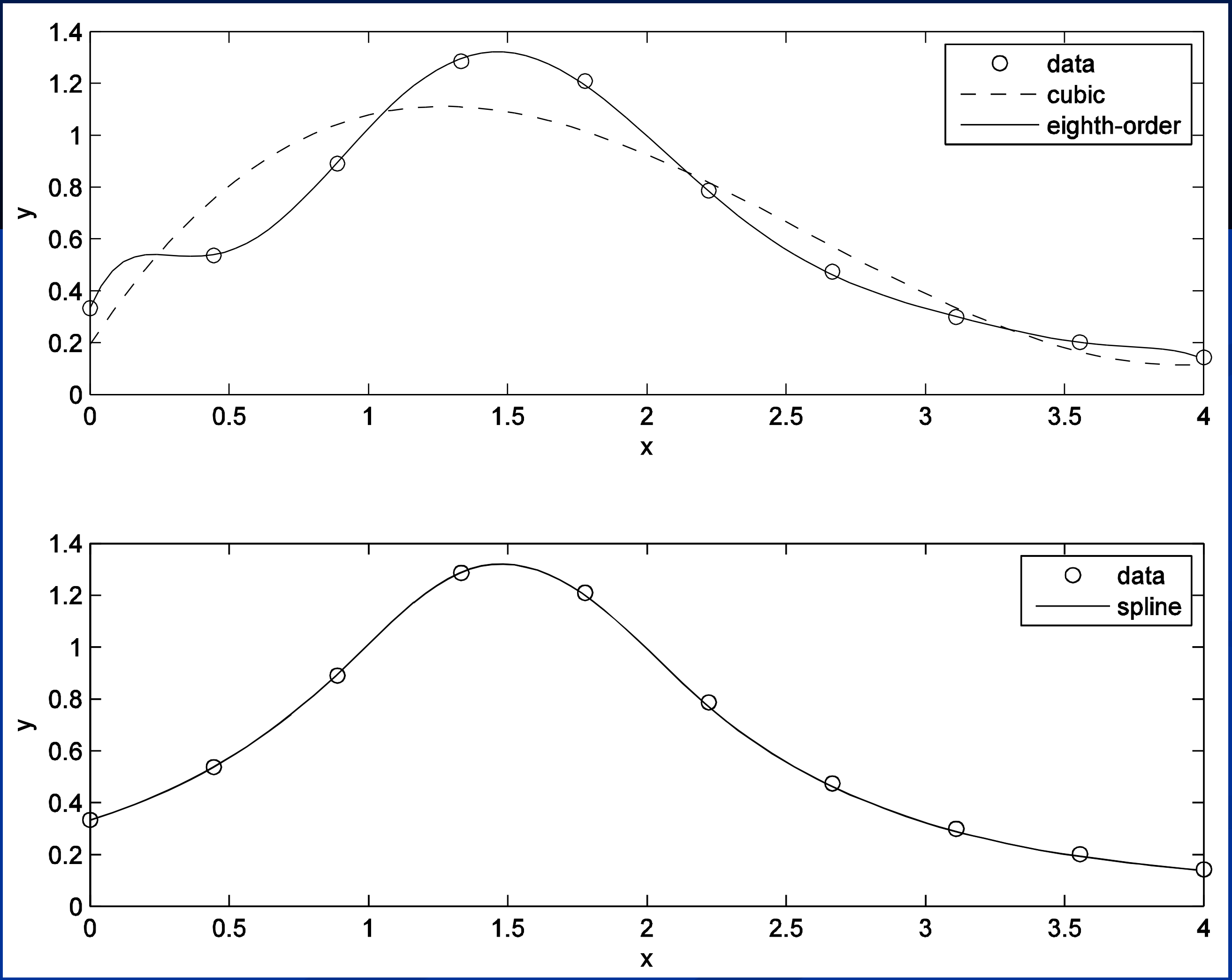
```
[breaks, coeffs, m, n] = unmkpp(spline(x,y))
```

Computes the coefficients of the cubic-spline polynomials for the data in `x` and `y`. The vector `breaks` contains the *x* values, and the matrix `coeffs` is an $m \times n$ matrix containing the polynomial coefficients. The scalars `m` and `n` give the dimensions of the matrix `coeffs`; `m` is the number of polynomials, and `n` is the number of coefficients for each polynomial.

The next slide illustrates interpolation using a cubic polynomial and an eighth order polynomial (top graph). The cubic is not satisfactory in this case, and the eighth order polynomial is not suitable for interpolation over the interval $0 < x < 0.5$.

The cubic spline does a better job in this case (bottom graph).

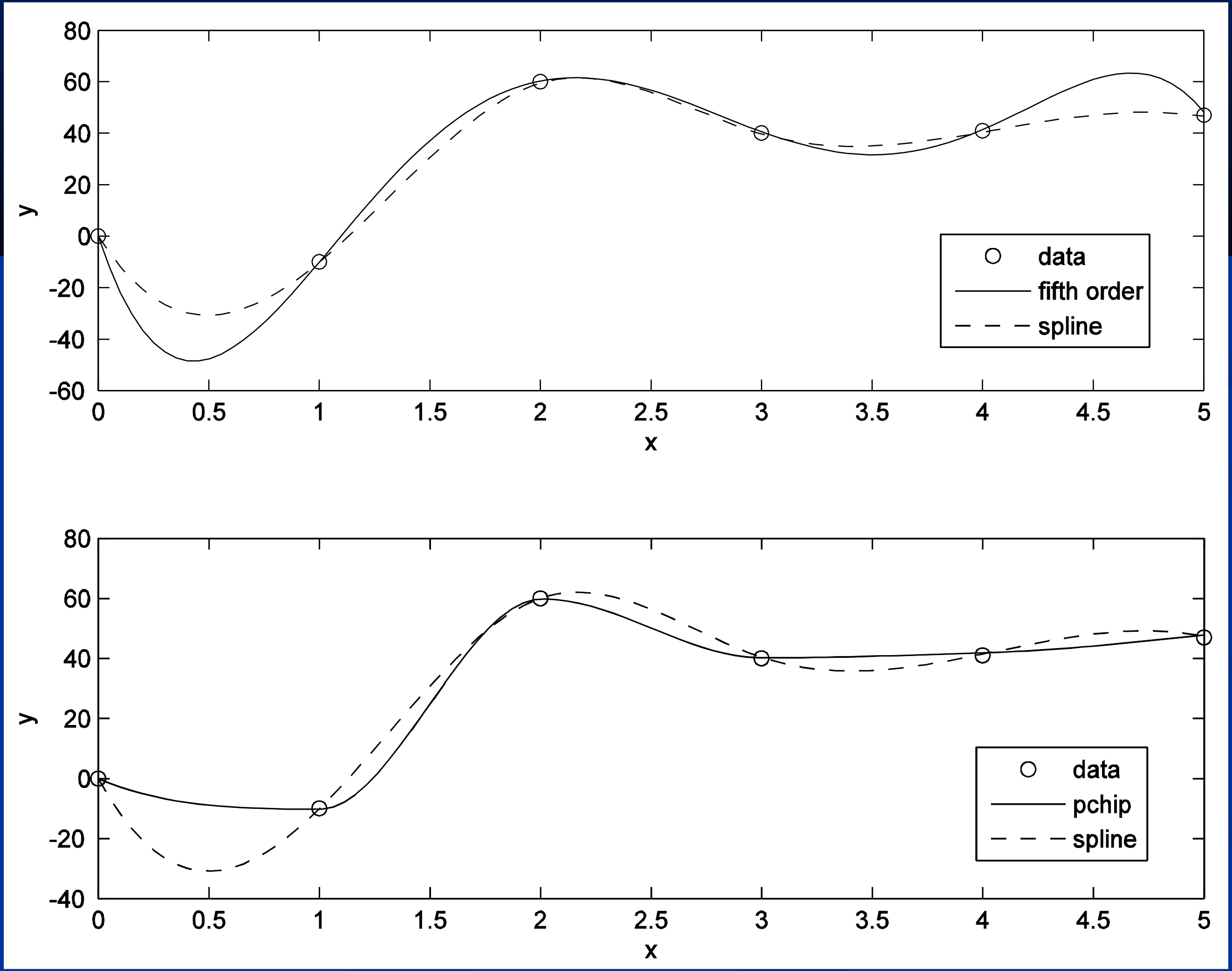
Figure 7.4-4 Top: Cubic and eighth order polynomial interpolation. Bottom: Cubic spline (page 321).



The next slide illustrates interpolation using a fifth order polynomial and a cubic spline (top graph). The cubic spline is better because the fifth order polynomial displays wide variations between the data points.

The pchip polynomial does a better job than the cubic spline in this case (bottom graph).

Figure 7.4-5 Top: Fifth order polynomial and cubic spline interpolation. Bottom: pchip and cubic spline interpolation. (page 323)


data
fifth order
spline
data
pchip
spline
x
y