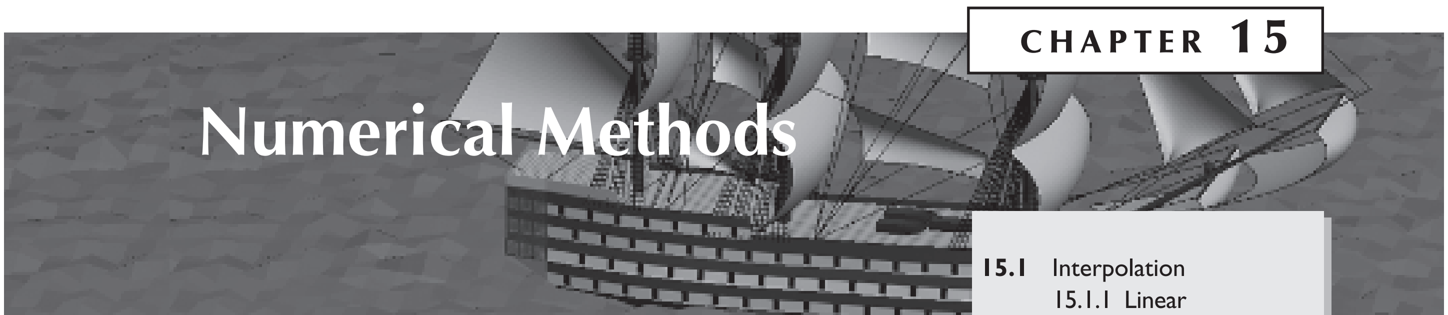


CHAPTER 15

Numerical Methods

Chapter Objectives

This chapter discusses the implementations of four common numerical techniques:

- Interpolating data
- Fitting polynomial curves to data
- Numerical integration
- Numerical differentiation

Introduction

Real-world data are rarely in such a form that you can use it immediately. Frequently, the data must be manipulated according to the user's actual needs:

- If the data samples have correct values but are not close enough together to be used directly, we can use interpolation to compute data points between the samples provided.
- There are occasions where the data-gathering facilities add some amount of noise to the data. To minimize the effects of the noise, we can compute the coefficients of a polynomial function that best matches the data.
- There are also times when the data must be integrated or differentiated to derive the quantities of interest.

15.1 Interpolation

If our data samples have correct values but are not close enough to be used directly, we can use either linear or cubic interpolation to compute data points between the samples provided. For example, plotting functions use linear interpolation to draw the lines between data points. In general, interpolation is a technique by which we estimate a variable's value between known values. In this section, we present the two most common types of interpolation: linear interpolation and cubic spline interpolation. In both techniques, we assume that we have a set of data points that represents *x*-*y* coordinates for which *y* is a function of *x*; that is,

```
y = f(x).
```

We then have a value of *x* that is not part of the data set for which we want to find the *y* value. Figure 15.1 illustrates the definition of the interpolation problem.

15.1.1 Linear Interpolation

Linear interpolation is one of the most common techniques for estimating data values between two given data points. With this technique, we assume that the function between the points can be represented by a straight line drawn between the points, as shown in Figure 15.2. Since we can find the equation of a straight line defined by the two known points, we can find *y* for any value of *x*. The closer the points are to each other, the more accurate our approximation is likely to be. Of course, we could use this equation to

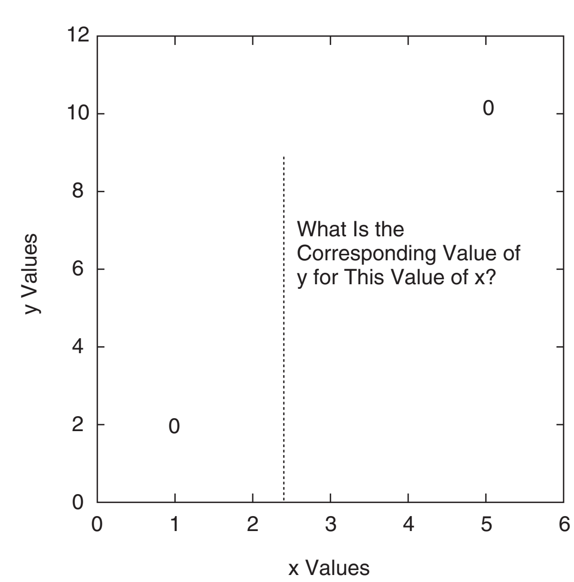


Figure 15.1 *The interpolation problem*

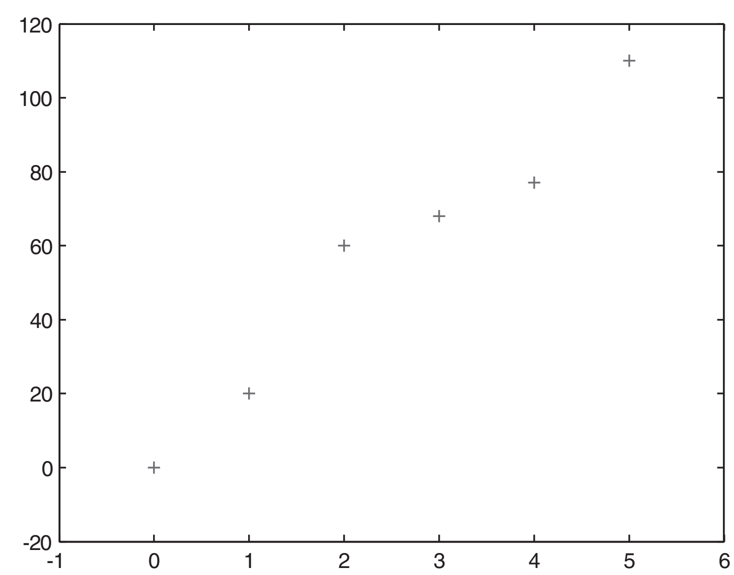


Figure 15.2 *Interpolation raw data*

extrapolate points past our collected data. This is rarely wise, however, and often leads to significant errors.

The function that performs linear interpolation is as follows:

```
new_y = interp1(x, y, new_x)
```

where the vectors `x` and `y` contain the original data values and the vector `new_x` contains the point(s) for which we want to compute interpolated `new_y` values. The `x` values should be in ascending order, and the `new_x` values should be within the range of the original `x` values. Note that the last character in the name `interp1` is the numeric 1 (one), not a lowercase L.

The use of `interp1(...)` is demonstrated in Listing 15.1.

Listing 15.1 Linear interpolation

```
x = 0:5;
y = [0, 20, 60, 68, 77, 110];
plot(x, y, 'r+')
hold on
fprintf('value at 1.5 is %2.2f\n', interp1(x, y ,1.5));
new_x = 0:0.241:5;
new_y = interp1(x,y,new_x);
plot(new_x, new_y, 'o')
axis([-1,6,-20,120])
title('linear Interpolation Plot')
xlabel('x values') ; ylabel('y values')
fprintf('value at 7 is %2.2f\n', interp1(x, y ,7));
```

In Listing 15.1:

Lines 1–3: We use the data illustrated in Figure 15.2.

Line 5: We take a single interpolated reading from the data at `x = 1.5`.

Lines 4–8: We plot points spaced 0.241 units apart on the x-axis marked with circles, as shown in Figure 15.3. Notice that the circles fall on the straight lines between the given data values.

Lines 9–11: Document the plot.

Line 12: Finally, we attempt to extrapolate to the point `x = 7` and see that `NaN (Not a Number)` is returned because `interp1` refuses to extrapolate outside the original range of x values. The output from running this script is:

```
value at 1.5 is 40.00
value at 7 is NaN
```

The MATLAB language allows us to provide a fourth parameter to the `interp1` function that must be a string that modifies its behavior. The choices are as follows:

`'nearest'` nearest neighbor interpolation

`'linear'` linear interpolation—the default

`'spline'` piecewise cubic spline interpolation (see Section 15.1.2)

`'pchip'` shape-preserving piecewise cubic interpolation

`'cubic'` same as `'pchip'`

`'v5cubic'` cubic interpolation that does not extrapolate, and uses `'spline'` if x is not equally spaced

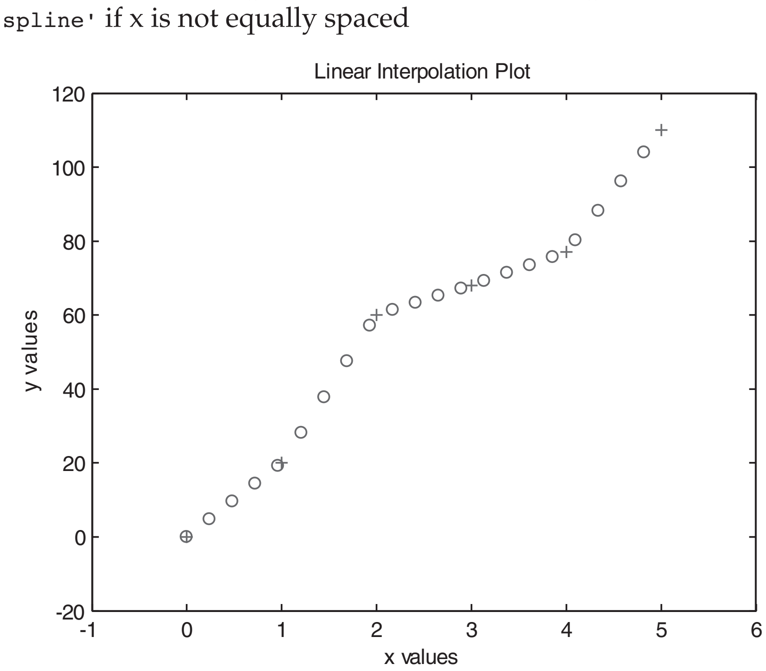


Figure 15.3 *Linear interpolation*

The MATLAB language also provides for two-dimensional (`interp2`) and three-dimensional (`interp3`) interpolation functions, which are not discussed here.

15.1.2 Cubic Spline Interpolation

A **cubic spline** is a smooth curve constructed to go through a set of points. The curve between each pair of points is a third-degree polynomial that has the general form:

$$x = a_{x0}t^3 + a_{x1}t^2\ 1\ a_{x2}t + a_{x3} \text{ and}$$
$$y = a_{y0}t^3 + a_{y1}t^2 + a_{y2}t + a_{y3}$$

where t is a parameter ranging from 0 to 1 between each pair of points. The coefficients are computed so that this provides a smooth curve between pairs of points and a smooth transition between the adjacent curves. Figure 15.4 shows a cubic spline smoothly connecting six points using a total of five different cubic equations.

The function that performs linear interpolation is as follows:

```
new_y = spline(x, y, new_x);
```

where the vectors `x` and `y` contain the original data values, and the vector `x_new` contains the point(s) for which we want to compute interpolated `y_new` values. The `x` values should be in ascending order, and while the `x_new` values should be within the range of the `x` values, this function will attempt to extrapolate outside that range.

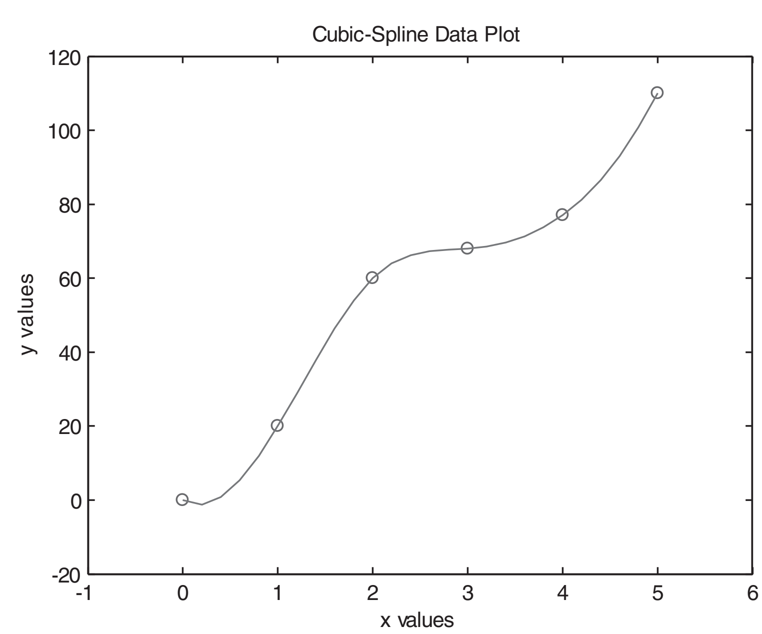


Figure 15.4 *Cubic spline interpolation*

Listing 15.2 Spline interpolation

```
1. x = 0:5;
2. y = [0, 20, 60, 68, 77, 110];
3. new_x = 0:0.2:5;
4. new_y = spline(x, y, new_x);
5. plot(x, y, 'o', new_x, new_y, '-')
6. axis([-1,6,-20,120])
7. title('Cubic-Spline Data Plot')
8. xlabel( 'x values'); ylabel('y values')
```

The curve in Figure 15.4 was created using the code shown in Listing 15.2.

Style Points 15.1

A good convention to adopt is shown in Figure 15.4:

- Use symbols to plot data points that are real values with no associated information connecting them
- Draw lines between data points only when there is an analytic relationship that connects the data points

Here, we use a circle symbol for the raw data to emphasize the original source of the information, and a smooth line for the spline curve to indicate that we are assuming a possibly erroneous but continuous relationship between data points.

In Listing 15.2:

Lines 1 and 2: Show the original `x` and `y` values.

Line 3: Shows dense `x` values to define the curve.

Line 4: Computes the spline function.

Lines 5–8: Plot the original data and the smooth curve.

15.1.3 Extrapolation

A note of caution about extrapolation—attempting to infer the values of data points outside the range of data provided is problematic at best and usually gives misleading results. Although logically your code may allow you to, you should never do it. The `interp1` and `spline` functions behave differently in this respect. As we saw previously, the `interp1` function refuses to supply results outside the range of the original `x` data. If you try, for every `new_x` value outside the range of the original `x` values, it will return `NaN`—not a number.

This is actually quite nice because if you accidentally request interpolated data like this, the plot programs ignore `NaN` values. The `spline` function, however, has no such scruples and allows you to request any `x` values you want, using the equation of the closest line segment. So considering Figure 15.4, if you asked for the value at `x = -3`, it would use the segment between 0 and 1, which has a violent upswing at the lower end (see Exercise 15.1).

Exercise 15.1 The evils of extrapolation

After running the script in Listing 15.1, enter this code:

```
>> spline(x, y, -3)
ans =
  813.3333
```

This might be what you want, but it looks odd! Chances are the data are not as accurate as you thought, and you probably need to fit a curve to the data, as explained in the following section.

15.2 Curve Fitting

There are occasions where the data acquisition facilities add some amount of noise to the data. To minimize the effects of the noise, we can smooth the data by computing the coefficients of a polynomial function that best match the data. The choice of the order of the polynomial must be made by the users, depending upon their understanding of the underlying physics that generated the data.

For example, assume that we have a set of data points collected from an experiment. After plotting the data points, we find that they generally fall in a straight line. However, if we were to try to draw a straight line through the points, probably only a couple of the points would fall exactly on the line. A least squares curve fitting method could be used to find the straight line that is the closest to the points, by minimizing the distance from each point to the straight line. Although this line can be considered a “best fit” to the data points, it is possible that none of the points would actually fall on the line of best fit. (Note that this method is different from interpolation, because the lines used in interpolation actually fall on all of the original data points.)

In the following section, we will discuss fitting a straight line to a set of data points, and then we will discuss fitting a polynomial of higher order.

15.2.1 Linear Regression

Linear regression is the process that determines the linear equation that is the best fit to a set of data points in terms of minimizing the sum of the squared distances between the line and the data points. To understand this process, first we consider the same set of data values used previously and attempt to “eyeball” a straight line through the data. Assume, for example, that $y = 20x$ is a good estimate of the curve. Listing 15.3 shows the code to plot the points and this estimate.

Listing 15.3 Eyeball linear estimation

```
x = 0:5;
y = [0 20 60 68 77 110];
y2 = 20 * x;
plot(x, y, 'o', x, y2);
axis([-1 7 -20 120])
title('Linear Estimate')
xlabel('Time (sec)')
ylabel('Temperature (degrees F)')
grid on
```

In Listing 15.3:

Lines 1 and 2: Show the original data points.

Line 3: Is our eyeball estimate.

Lines 4–9: Plot the original data and the estimate.

Looking at the results in Figure 15.5, it appears that $y = 20x$ is a reasonable estimate of a line through the points. We really need the ability to compare the quality of the fit of this line to other possible estimates, so we compute the difference between the actual y value and the value calculated from the estimate:

```
>> dy = [0, 0, 20, 8, -3, 10]
```

It turns out that the best way to make this comparison is by the **least squares technique,** whereby the measure of the quality of the fit is the sum of the squared differences between the actual data points and the linear estimates. This sum can be computed with the following command:

```
>> sum_sq = sum(dy.^2)
```

For the above set of data, the value of `sum_sq` is `573`. As we will see, MATLAB can automatically produce the best linear fit shown in Figure 15.6 whose sum of squares is `356.82`, a significant improvement over our original guess. This result was achieved by running Exercise 15.2.

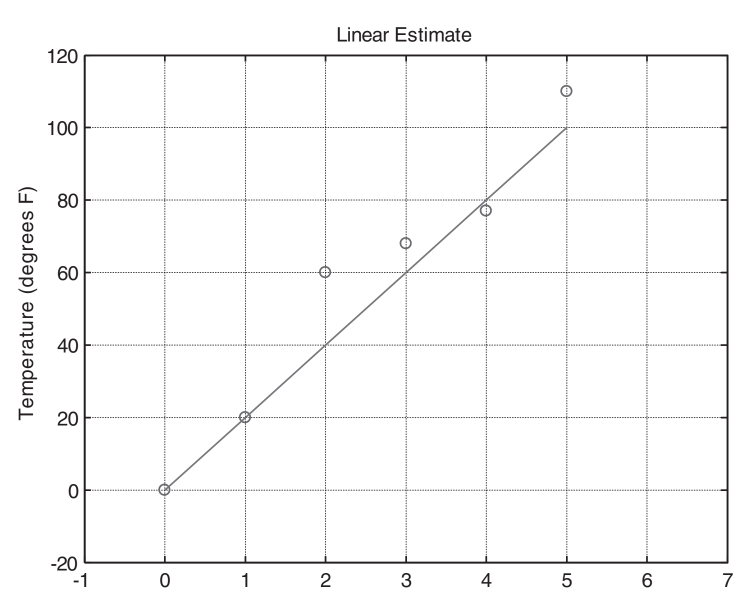


Figure 15.5 *An eyeball estimate of a linear fit*

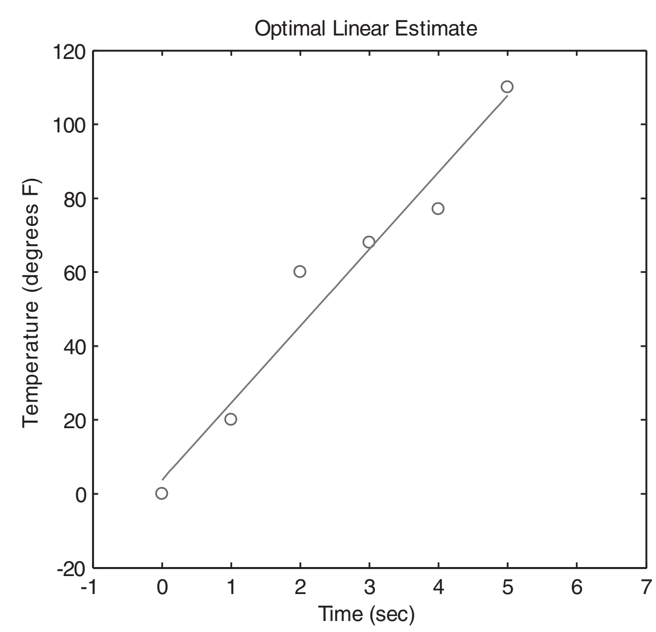


Figure 15.6 *Linear curve fit*

Exercise 15.2 Optimal linear fit

Again using the data from Section 15.1.1:

```
>> x=0:5; y=[0,20,60,68,77,110]
>> polyfit(x, y, 1)
ans =
      20.8286  3.7619
```

15.2.2 Polynomial Regression

Linear regression is a special case of the polynomial regression technique. Recall that a polynomial with one variable can be written by using the following formula:

$$f(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + a_3x^{n-3} + \dots a_{n-1}x + a_n$$

The degree of a polynomial is equal to the largest value used as an exponent. MATLAB provides a pair of functions to compute the coefficients of the best fit to a set of data and then interpolate on those coefficients to produce the data to plot:

- `coef = polyfit(x, y, n)` computes the coefficients of the polynomial of degree `n` that best matches the given `x` and `y` values. The function returns the coefficients, `coef`, in descending powers of `x`. For the least squares calculation to work, the length of `x` should be greater than `n - 1`. If this is not the case, the coefficients are still computed, but the curve passes through all the data points.

- `new_y = polyval(coef, new_x)` can then be used to interpolate the polynomial defined by these coefficients for the `new_y` value(s) corresponding to any `new_x` value(s).

Note that there is nothing to prevent you from using these coefficients for extrapolation.

Exercise 15.2 illustrates fitting the best straight line to the data used in Section 15.1.1, indicating that the first-order polynomial that best fits our data is as follows:

$$f(x) = 20.8286x + 3.7169$$

We could interpolate the values of `new_x` with:

```
new_y = coef(1) * new_x + coef(2)
```

or we could use the function `polyval`:

```
new_y = polyfit(coef, new_x)
```

We can use our new understanding of the `polyfit` and `polyval` functions to write a program to study the improvement in the curve fit as `n` increases, as shown in Listing 15.4.

In Listing 15.4:

Lines 1–3: Set up the data sets.

Lines 4–14: Study second- through fifth-order fits.

Line 5: Combines `polyfit` and `polyval` calls to compute the new y values.

Lines 6–12: Plot the results. Notice the use of `sprintf(...)` to make a dynamic title for the plots.

Listing 15.4 Higher-order fits

```
 1. x = 0:5;
 2. fine_x = 0:.1:5;
 3. y = [0 20 60 68 77 110];
 4. for order = 2:5
 5.     y2=polyval(polyfit(x,y,order), fine_x);
 6.     subplot(2,2,order-1)
 7.     plot(x, y, 'o', fine_x, y2)
 8.     axis([-1 7 -20 120])
 9.     ttl = sprintf('Degree %d Polynomial Fit', ...
10.                   order );
11.     title(ttl)
12.     xlabel('Time (sec)')
13.     ylabel('Temperature (degrees F)')
14. end
```

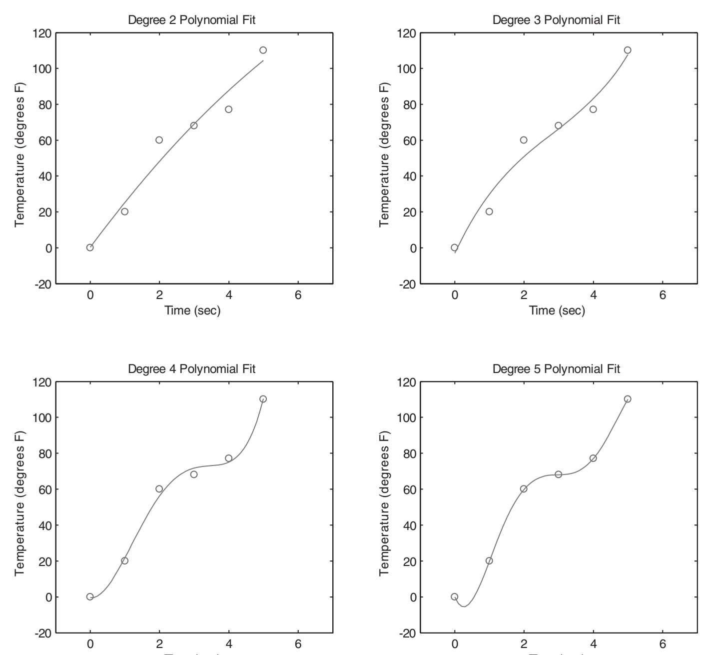


Figure 15.7 *Higher-order polynomial fits*

The results are shown in Figure 15.7. Notice that with six points, the fifth-order fit goes through all the data points.

15.2.3 Practical Application

We return briefly to the problem of replacing the blue sky in Chapter 13. The sky we used to replace the gray skies of Vienna has a power line we need to remove. We can use polynomial curve fitting to create an artificial sky with exactly the same color characteristics as the blue sky in the cottage picture, but without the wire. This is possible because each row of the image has so much data that define its color profile that the presence of the wire is a minor amount of "noise." We merely need to process each row of the sky, fit a second-order curve to it, interpolate a new sky row from the parameters, and replace the row in the sky. The code to perform this is shown in Listing 15.5.

In Listing 15.5:

Line 1: Reads the original cottage picture.
Line 2: Obtains its sizes.
Line 3: The x values for the curve fitting.
Line 4: Makes a copy of the original picture.
Lines 5–12: Convert the top 700 rows where the sky is.
Lines 6–11: Treat each color individually.

Listing 15.5 Removing the power line from the sky

```
 1. p = imread('Witney.jpg');
 2. [rows, cols, clrs] = size(p);
 3. x = 1:cols;
 4. sky = p;
 5. for row = 1:700
 6.     for color = 1:3
 7.         cv = double(p(row, :, color));
 8.         coef = polyfit(x, cv, 2);
 9.         ncr = polyval(coef, x);
10.         sky(row,:,color) = uint8(ncr);
11.     end
12. end
13. image(sky)
14. imwrite(sky, 'sky.jpg');
```

Line 7: The polynomial approximation needs each row as a double vector.

Lines 8–9: Compute a synthetic row.

Line 10: Puts the row into the new sky.

Lines 13 and 14: Show and save the new image.

Figure 15.8 shows the cottage picture updated with a smooth sky. Notice that the chimneys have been smeared off, but this does not affect the part of the sky needed for the Vienna picture. This synthetic sky is ready to be used in the script to replace the original sky (see Listing 13.1). Figure 15.9 shows the Vienna picture with a clear blue synthetic sky.

Figure 15.8 *Updated sky*

Figure 15.9 *Updated picture*

15.3 Numerical Integration

The integral of a function $f(x)$ over the interval $[a, b]$ is defined to be the area under the curve of $f(x)$ between a and b, as shown in Figure 15.10. If the value of this integral is K, the notation to represent the integral of $f(x)$ between a and b is as follows:

$$K = \int_a^b f(x)\, dx$$

For many functions, this integral can be computed analytically. However, for a number of functions, this is not possible, and we require a numerical technique to estimate its value. We look at two different scenarios:

- Two different techniques for computing the complete integral with various degrees of accuracy
- A technique for evaluating the continuous integral of $f(x)$

15.3.1 Determination of the Complete Integral

Two of the most common numerical integration techniques estimate $f(x)$ either with a set of piecewise linear functions or with a set of piecewise parabolic functions. If we use piecewise linear functions, we can compute the area of the trapezoids that compose the area under the piecewise linear function. This technique is called the **trapezoidal rule.** If we use piecewise quadratic functions, we can compute and add the areas of these components. This technique is called **Simpson's rule.**

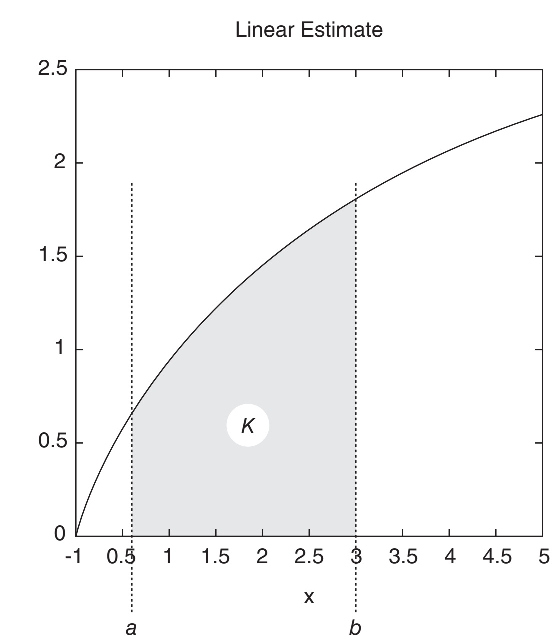


Figure 15.10 *Integration of f(x)*

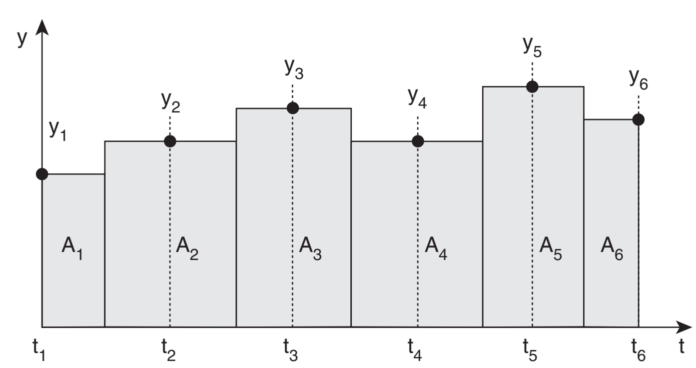


Figure 15.11 *Discrete integration*

The Trapezoidal Rule If we represent the area under a curve by trapezoids, as illustrated in Figure 15.11, and if the interval $[a, b]$ is divided into n equal sections, then the area can be approximated by the following formula:

$$K_T = \frac{b - a}{2n}\left(f(x_0) + 2f(x_1) + 2f(x_2) + \ \ldots\ + 2f(x_{n-1}) + f(x_n)\right)$$

where the x_i values represent the end points of the trapezoids and where $x_0=a$ and $x_n=b$. Listing 15.6 shows a function that computes this integral.

Simpson's Rule If the area under a curve is represented by areas under quadratic sections of a curve, and if the interval $[a, b]$ is divided into

Listing 15.6 Trapezoidal integration

```
function K = trapezoid( v, a, b )
% usage: K = trapezoid(v, a, b )
K = (b-a) * (v(1) + v(end) + ...
2*sum(v(2:end-1))) / (2*(length(v) - 1) );
```

$2n$ equal sections, then the area can be approximated by the formula (Simpson's rule):

$$K_s = \frac{h}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ \ldots \right.$$
$$\left. + \ 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})\right)$$

where the x_i values represent the end points of the sections, $x_0 = a$ and $x_{2n} = b$, and $h = (b - a)\ /\ (2n)$. Listing 15.7 shows a function to integrate using Simpson's rule.

15.3.2 Continuous Integration Problems

We now consider a slightly different scenario. If $f(t)$ is the rate of change of $F(t)$ defined as $f(t) = dF(t)/dt$, then given $f(t)$, we can find the indefinite integral $F(t)$ according to the following formula:

$$F(t) = \int_{t_0}^{t} f(x)\, dt$$

For example, we might be given data that represent the velocity of a sounding rocket, such as is plotted in Figure 15.12. We need to approximate the altitude of the rocket over time by integrating this data.

To perform this kind of integral, the MATLAB language provides the function F = `cumsum(f)` that computes the cumulative sum of the vector `f`. The result, `F`, is a vector of the same length as f where F`(i)` is the sum of f`(1:i)`. If the data values, `f`, are regularly sampled at a rate Δ`t`, the integral is found by multiplying `cumsum(f)` by the time interval, Δ`t`. If they are not regularly sampled, you have to compute the `cumsum(...)` of the scalar product of `f` and the vector of time differences.

Listing 15.7 Simpson's rule integration

```
function K = simpson( v, a, b )
% usage: K = simpson(v, a, b )
K = (b-a) * (v(1) + v(end) + ...
        4*sum(v(2:2:end-1)) + ...
        2*sum(v(3:2:end-2)) ) / (3*(length(v) - 1) );
```

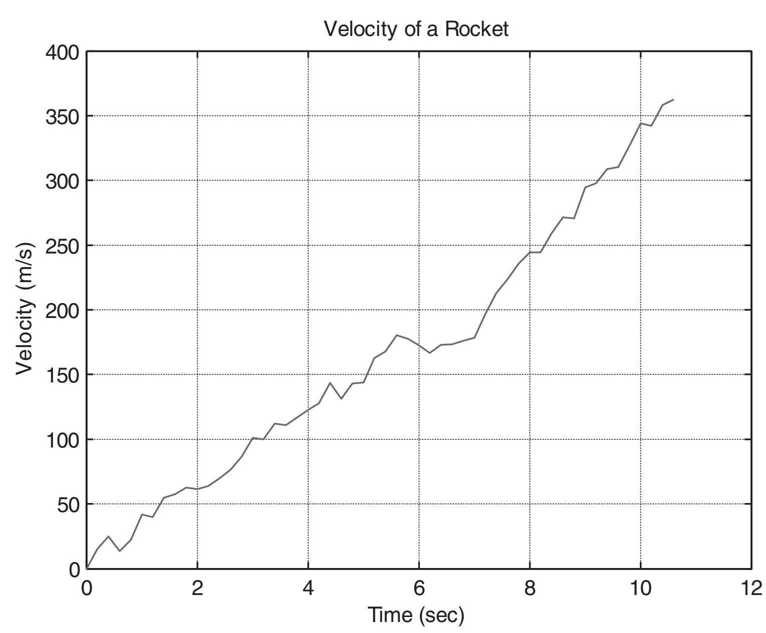


Figure 15.12 *Velocity of a rocket*

To compute a more accurate integral, especially if the samples are not regularly spaced along the independent axis, MATLAB also provides the function `cumtrapz(t, f)` where `t` is the independent parameter and `f` the dependent parameter. The function uses trapezoidal integration to calculate the indefinite integral `F(t)`.

Listing 15.8 shows the function that computes this continuous integral, making use of `cumsum(...)`.

Listing 15.8 Integrating rocket velocity

```
v =[ 0.0 15.1 25.1 13.7 22.2 41.7 ...
     39.8 54.8 57.6 62.6 61.6 63.9 69.6 ...
     76.2 86.7 101.2 99.8 112.2 111.0 ...
    116.8 122.6 127.7 143.4 131.3 143.0 ...
    144.0 162.7 167.8 180.3 177.6 172.6 ...
    166.6 173.1 173.3 176.0 178.5 ...
    196.5 213.0 223.6 235.9 244.2 244.5 ...
    259.4 271.4 270.5 294.5 297.6 ...
    308.7 310.5 326.6 344.1 342.0 358.2 362.7 ];
lv = length(v);
dt = 0.2;
t = (0:lv-1) * dt;
h = dt * cumsum(v);
plot(t, v)
hold on
plot(t, h/5,'k--')
legend({'velocity', 'altitude/5' })
```

continued on next page

```
18. title('velocity and altitude of a rocket')
19. xlabel('time (sec)'); ylabel('v (m/s), h(m/5)')
20. fprintf('cumsum height: %g\n', h(end) );
21. fprintf('trapezoidal height: %g\n', ...
22.           trapezoid(v, t(1), t(end) ));
23. fprintf('Simpson''s Rule height: %g\n', ...
24.           simpson(v, t(1), t(end) ));
```

In Listing 15.8:

Lines 1–9: Generate the original velocity data.

Lines 10–12: Parameters for plotting.

Line 13: Performs the integration.

Lines 14–19: Plot the results.

Lines 20–24: Validate the three integration techniques by checking the results.

Figure 15.13 shows the resulting plot. The results displayed in the Command window are:

```
cumsum height: 1848.5
trapezoidal height: 1811.85
Simpson's Rule height: 1811.14
```

The continuous integration produces results within 2% of the "accurate" integration techniques.

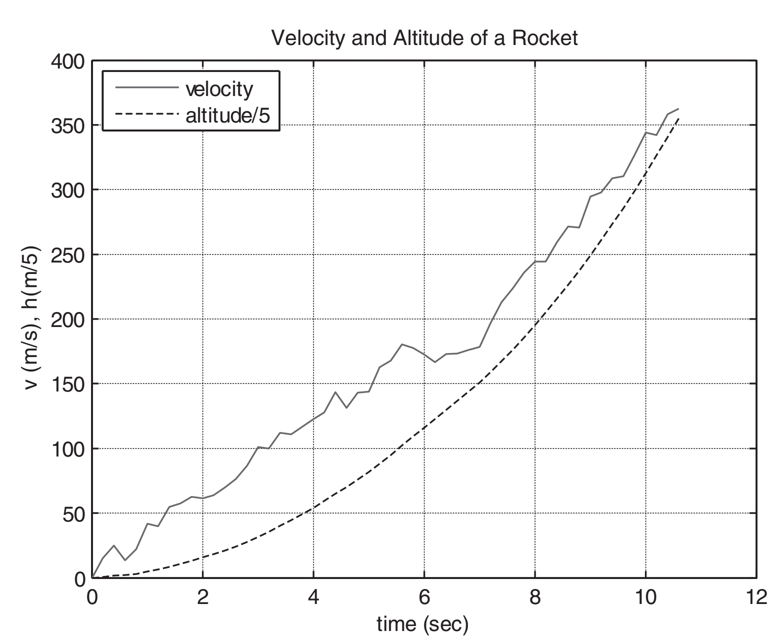


Figure 15.13 *Rocket velocity and altitude*

15.4 Numerical Differentiation

The derivative of a function $f(x)$ is defined to be a function $f'(x)$ that is equal to the rate of change of $f(x)$ with respect to x. The derivative can be expressed as a ratio, with the change in $f(x)$ indicated by $df(x)$ and the change in x indicated by dx, giving us the following:

$$f'(x) = \frac{df(x)}{dx}$$

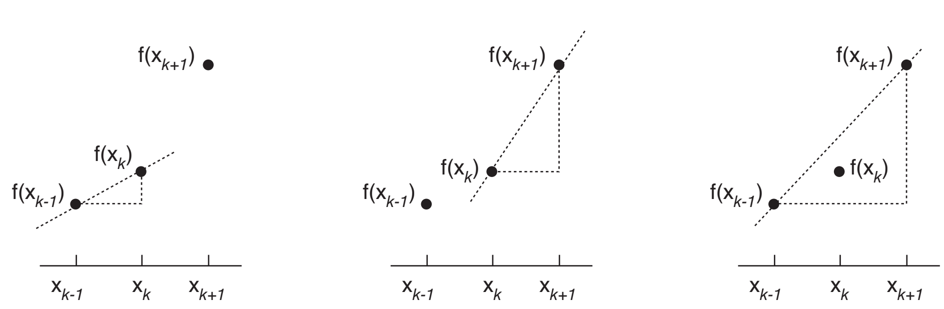
There are many physical processes for which we want to measure the rate of change of a variable. For example, velocity is the rate of change of position (as in meters per second), and acceleration is the rate of change of velocity (as in meters per second squared).

The derivative $f'(x)$ can be described graphically as the slope of the function $f(x)$, which is defined to be the slope of the tangent line to the function at the specified point. Thus, the value of $f'(x)$ at the point a is $f'(a)$, and it is equal to the slope of the tangent line at the point a.

15.4.1 Difference Expressions

In general, numerical differentiation techniques estimate the derivative of a function at a point x_k by approximating the slope of the tangent line at x_k using values of the function at points near x_k. The approximation of the slope of the tangent line can be done in several ways, as shown in Figure 15.14.

- *Backward Difference:* Figure 15.14(a) assumes that the derivative at x_k is estimated by computing the slope of the line between $f(x_{k-1})$ and $f(x_k)$
- *Forward Difference:* Figure 15.14(b) assumes that the derivative at x_k is estimated by computing the slope of the line between $f(x_k)$ and $f(x_{k+1})$



(a) Forward (b) Backward (c) Central

Figure 15.14 *Difference techniques*

- *Central Difference:* Figure 15.14(c) assumes that the derivative at x_k is estimated by computing the slope of the line between $f(x_{k-1})$ and $f(x_{k+1})$

The quality of all of these types of derivative computations depends on the distance between the points used to estimate the derivative; the estimate of the derivative improves as the distance between the two points decreases.

15.5 Analytical Operations

We return to the discussion of fitting a polynomial to some raw data in Section 15.2.2. We approximated a polynomial fit with the following expression:

$$f(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + a_3x^{n-3} + \ldots a_{n-1}x + a_n$$

Since this is an analytical expression, even if some or all of the coefficients are complex, we can integrate it to estimate the integral of the raw data and differentiate it to estimate the slope of the raw data.

15.5.1 Analytical Integration

The expression for F(x), the integral of f(x) with respect to x, is given by:

$$F(x) = a_0x^{n+1}/(n+1) + a_1x^n/n + a_2x^{n-1}/(n-1) + a_3x^{n-2}/(n-2) + \ldots$$
$$a_{n-1}x^2/2 + a_n\, x + K$$

Note that an arbitrary constant, K, is always required for analytical integration representing the starting value F(0).

15.5.2 Analytical Differentiation

The expression for f′(x), the integral of f(x) with respect to x, is given by:

$$f'(x) = na_0x^{n-1} + (n-1)a_1x^{n-2} + (n-2)a_2x^{n-3} + (n-3)a_3x^{n-4} + \ldots a_{n-1}$$

15.6 Implementation

To facilitate differentiation, the MATLAB language defines the `diff(...)` function, which computes differences between adjacent values in a vector, generating a new vector with one less value than the original:

```
dv = diff(V) returns [V(2)-V(1), V(3)-V(2), ..., V(n)-V(n-1)]
```

An approximate derivative dy/dx can be computed by using `diff(y)./diff(x)`. Depending on the application, this can be used to compute the

Listing 15.9 Differentiating a function

```
1. x = -7:0.1:9;
2. f = polyval([0.0333,-0.3,-1.3333,16,0,-187.2,0], x);
3. plot(x, f)
4. hold on
5. df = diff(f)./diff(x);
6. plot(x(2:end), df, 'g')
7. plot(x(1:end-1), df, 'r')
8. xm = (x(2:end)+x(2:end)) / 2
9. plot(xm, df, 'c')
10. grid on
11. legend({'f(x)', 'forward', 'backward', 'central'})
```

forward, backward, or central difference approximation. The solution to the forward difference is shown in Listing 15.9.

In Listing 15.9:

Lines 1–4: Establish and plot $f(x)$.

Line 5: The difference expression—returns a vector one shorter than the original.

Lines 6–11: Plot the forward, backward, and central differences.

The results are shown in Figure 15.15. Since the original data were generated from a series of coefficients, we could also plot the exact value of the slope using the result of Section 15.5.2.

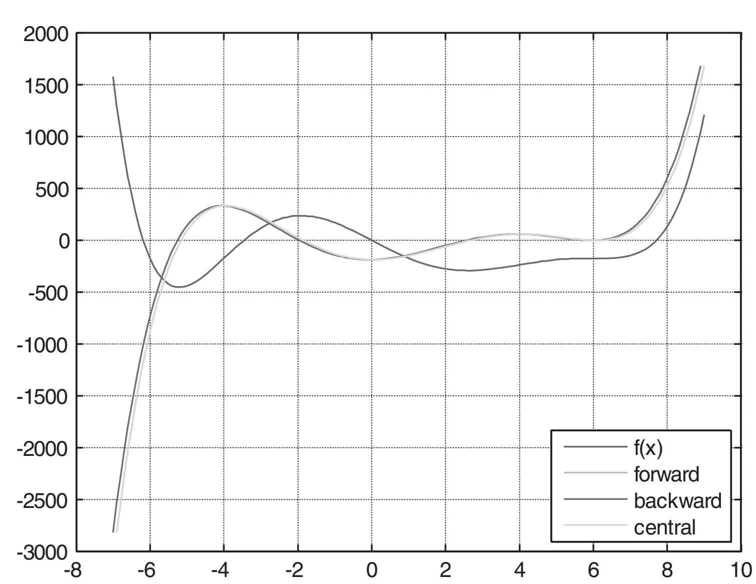


Figure 15.15 *Differentiation*

15.7 Engineering Example—Shaping the Synthesizer Notes

As discussed in Chapter 14, we can synthesize the frequency content of an instrument by selecting an appropriate number of coefficients from the energy spectrum, multiplying each by an appropriate sine or cosine wave and summing the results. This gives a time trace with constant amplitude, which is fine for an instrument like a trumpet, but notes played on other instruments like a piano have a very non-linear time profile as shown in Figure 15.16. That same figure has two overlays indicating how to develop the decay profile typical of a piano note. First, we choose a

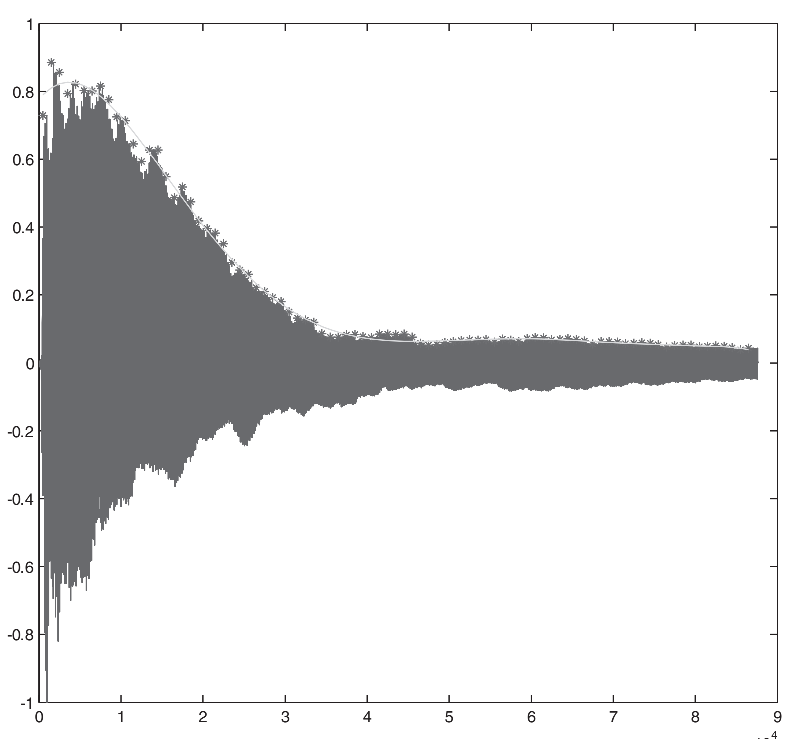


Figure 15.16 *Piano note time history*

modest number of samples that describe the envelope of the sound (marked by red * symbols). Then, we perform a high-order curve fit on that data and check its accuracy by plotting it as the solid line on the same figure.

To demonstrate the essence of this capability, we begin with Listing 14.6 from Chapter 14, reading the file 'instr_piano.wav' instead of 'instr_tpt.wav.' Now, we insert the code in Listing 15.10 in place of the last two lines of Listing 14.6.

Listing 15.10 Modifying sound amplitude

```
 1. figure
 2. plot(snd)
 3. hold on
 4. incr = 1000;
 5. at = 1;
 6. samples = [];
 7. tm = [];
 8. while at < (N - incr)
 9.    val = max(snd(at:at+incr-1));
10.    samples = [samples val];
11.    tm = [tm at+incr/2];
12.    at = at + incr;
13. end
14. plot(tm, samples,'r*')
15. coeff = polyfit(tm, samples, 8);
16. samp = polyval(coeff, tm);
17. plot(tm, samp, 'c')
18. amult = polyval(coeff, 1:length(f));
19. f = f .* amult;
20. sf = f ./ max(f);
21. sound(sf, Fs)
```

In Listing 15.10:

Lines 1–3: Plot the sound's time history in new figure window.

Line 4: Arbitrarily choose a time sample increment to achieve a small but representative set of amplitude samples.

Lines 8–13: A loop to compute and store the amplitude samples and the corresponding time indices. Each step calculates the maximum amplitude during its time window and saves it with the window location.

Line 14: Plots the sample locations.

Lines 15–17: Compute and plot the polynomial fit to the amplitudes using an eight-order fit.

Lines 18–21: Modify the synthesized piano sound by multiplying by the amplitude profile determined in this script.

In conclusion, with these two engineering examples, we have shown how the essence of the sound of a musical instrument can be derived from the actual sound of an instrument and captured as a small set of complex amplitudes with their frequency value and an even smaller set of real coefficients of the function that multiplies the amplitude over time.

To construct from these data a real music synthesizer, one need only to detect that a keyboard note has been pressed, determine the required frequency, and play the synthesized note until the key is released. If the synthesizer is equipped to specify that the sustain pedal is depressed, the piano sound should not be cut off, but allowed to fade into silence.

Chapter Summary

In this chapter, we saw the implementations of four common numerical techniques:

- We can estimate data points between given data values using linear (`interp1/2/3`) or spline interpolation
- We can smooth noisy data by fitting polynomial curves of suitable order to the raw data
- Given, for example, the velocity of an object over time, we can determine its position by integrating using `cumtrapz(...)` or `cumsum(...)`
- We can differentiate to generate its acceleration

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`NaN`	Not a number	15.1.1
`cumsum(y)`	Computes the integral of the function y(x), assuming that Δx is 1	15.3.2
`cumtrapz(x,y)`	Computes the integral of the function y(x), using the trapezoidal rule	15.3.2
`diff(v)`	Computes the differences between adjacent values in a vector	15.6
`interp1(x, y, nx)`	Computes linear and cubic interpolation	15.1.1
`interp2(x, y, z, nx, ny)`	Computes linear and cubic interpolation	15.1.1
`interp3(x, y, z, v, nx, ny, nz)`	Computes linear and cubic interpolation	15.1.1
`polyfit(x, y, n)`	Computes a least-squares polynomial	15.2.2
`polyval(c, x)`	Evaluates a polynomial	15.2.2
`spline(x, y)`	Spline interpolation	15.1.2

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. All MATLAB functions permit extrapolation beyond the limits of the original independent variable.

2. The cubic spline is a series of parametric curves.
3. You cannot extrapolate the equations generated by curve fitting.
4. You should always match the order of a parametric curve fit to the underlying physics of the data.
5. Simpson's rule is more accurate than the trapezoidal rule for integrating a function.
6. Numerical differentiation produces a vector that is the same length as the original vector.

Fill in the Blanks

1. ____________ is the technique by which we estimate a variable's value between known values.
2. Nth-order polynomial regression determines the ____________ of order n that minimize the ______________ between the line and the data points.
3. The ___________________makes the slope at x_k the ______________ of the line between x_{k-1} and x_{k+1}.
4. To compute the continuous integral of a data set that is not regularly sampled, you have to compute the _________________ of the _______________ of ________________ and __________________.
5. If a(n) ____________ is defined by its polynomial coefficients, you can integrate or differentiate it by _______________ the vector of coefficients.

Programming Projects

1. Do the following basic exercises with numerical methods.
   a. Define two vectors `xi` and `yi` of the same length where the `xi` values are monotonically increasing and the `yi` values are somehow related to the `xi` values. Then define a new vector `x` with closer spacing than `xi` and extending below and above the range of `xi`. Find the `y` values corresponding to the `x` values in `xi` by linear interpolation. On the same figure, plot the original `yi` vs. `xi` as red circles, and `y` vs. `x` as a black line. What do you observe about the visible range of the `x` values?
   b. Repeat the exercise in part a using the `spline(...)` function to interpolate. Explain the difference in the range of the resulting `y` vs. `x` plot.
   c. Use `polyfit(...)` to find the coefficients of the third-order polynomial that best fits the points represented by vectors `xi` and

yi and then use polyval(...) to evaluate that curve at the x points. As before, plot yi vs. xi as red circles and y vs. x as a black line.

d. Approximate the derivative, dxy = dy/dx, for the vectors xi and yi using the diff(...) function and plot yi vs. xi. Since diff(...) reduces the length of the vector by one, you will have to plot dxy vs. either xi(1:end-1), xi(2:end) or compute xm, the mid-points of xi.

e. Find yp, the cumulative sum of the elements in dxy, and add this to the plot of part d. With the exception of a constant offset, this curve ought to track the original plot of yi vs. xi.

f. Use cumtrapz to find the area under the curve represented by yp vs. xi with the trapezoidal method of approximation. Compare this result to the ending value of the yp curve.

2. Write a function, bestFit, that takes in a vector of x-coordinates and a vector of y-coordinates. Your function should fit a polynomial curve to the data. The degree of the polynomial should be the smallest degree polynomial with an average error (the average of the absolute value of the difference between the new y-coordinates and the original y-coordinates) less than 2. Your function should return:
   - the vector of coefficients of your polynomial
   - the vector of new y-coordinates, which is the polynomial evaluated at the original x-coordinates, and
   - the vector of the error magnitudes of your polynomial.

   Write a test program to provide reasonable data to your function and plot the original data (in blue), the curve-fitted data (in green), and the error (in red) on one figure. Title your plot and label your axes accordingly, including a legend.

3. You have been approached by the Rambling Wreck club to test the performance of the Rambling Wreck. You are provided with the test results of the car for 10 trial runs in the form of a vector d that contains the displacement of the car from the origin at that second. The first element is the displacement at the 0th second, the second element is the displacement at the 1st second, and so on. Write a script called testWreck that displays a plot of the speed of the Rambling Wreck over time during the test run. You could test your script using:

```
d = [0 20 35 50 60 55 30 25 15 5];
```

4. Engineers often use tabulated data for various calculations. An important method that any good engineer should be able to apply to tabulated data is interpolation. In thermodynamics, the properties of a gas can be known when two of its properties have been fixed.

You are required to come up with a continuous function being given the tabulated data below measured where the pressure is 0.10 MPa:

Temperature (deg C)	Specific Volume (cu meters/Kg)
99.63	1.694
100	1.696
120	1.793
160	1.984
200	2.172
240	2.359
280	2.546
320	2.732
360	2.917
400	3.103
440	3.288
500	3.565

Write a function called lookup that consumes three parameters: the above table in an array, a number `value` , and a logical control value `getTemp`. If `getTemp` is `true`, the function interpolates the value as a specific volume and returns the corresponding temperature. Otherwise, it interpolates the value as a temperature and returns the corresponding specific volume. Your function must not extrapolate the data (i.e., it should return `NaN` if the user tries to obtain values outside the range of the table values).

5. Mathematically speaking, a critical point occurs when the derivative of a function equals zero. It is possible that a local minimum or a local maximum occurs at a critical point. A local minimum is a point where the function's value to the left and right of it is larger, and a local maximum is a point where the function's value to the left and right of it is smaller. You are going to write a function that finds the local minimum and maximum points of a set of data. Call the function `find_points`. It should take in vectors of `x and y` values and return two vectors. The first vector should contain the `x` values where the minimum points occur, while the second vector should contain the `x` values where the maximum points occur.

For example:

```
If x=linspace(-8,2,1000) and y=x.^2+6*x+3;
 [min_p max_p]=find_points(x,y) should return
       min_p = -3, max_p = []
If x=linspace(-5,5,1000) and y=x.^3-12*x;
 [min_p max_p]=find_points(x,y) should return:
       min_p = 2, max_p = -2
```

You should plot `x` and `y` to confirm the answers.

6. Now that we used the derivative it only makes sense that you are going to write a function that finds integrals. Call your function `find_integral`. Your function should take in a vector of `x` and `y` values as Problem 15.5 does and should plot the integral and also return the total area under the function. You are to use the trapezoidal rule to find the integrals.

   For example:

   ```
   If x=linspace(0,5,1000); and y=2*x+5;
   find_integral(x,y) should return 50.0000
   ```