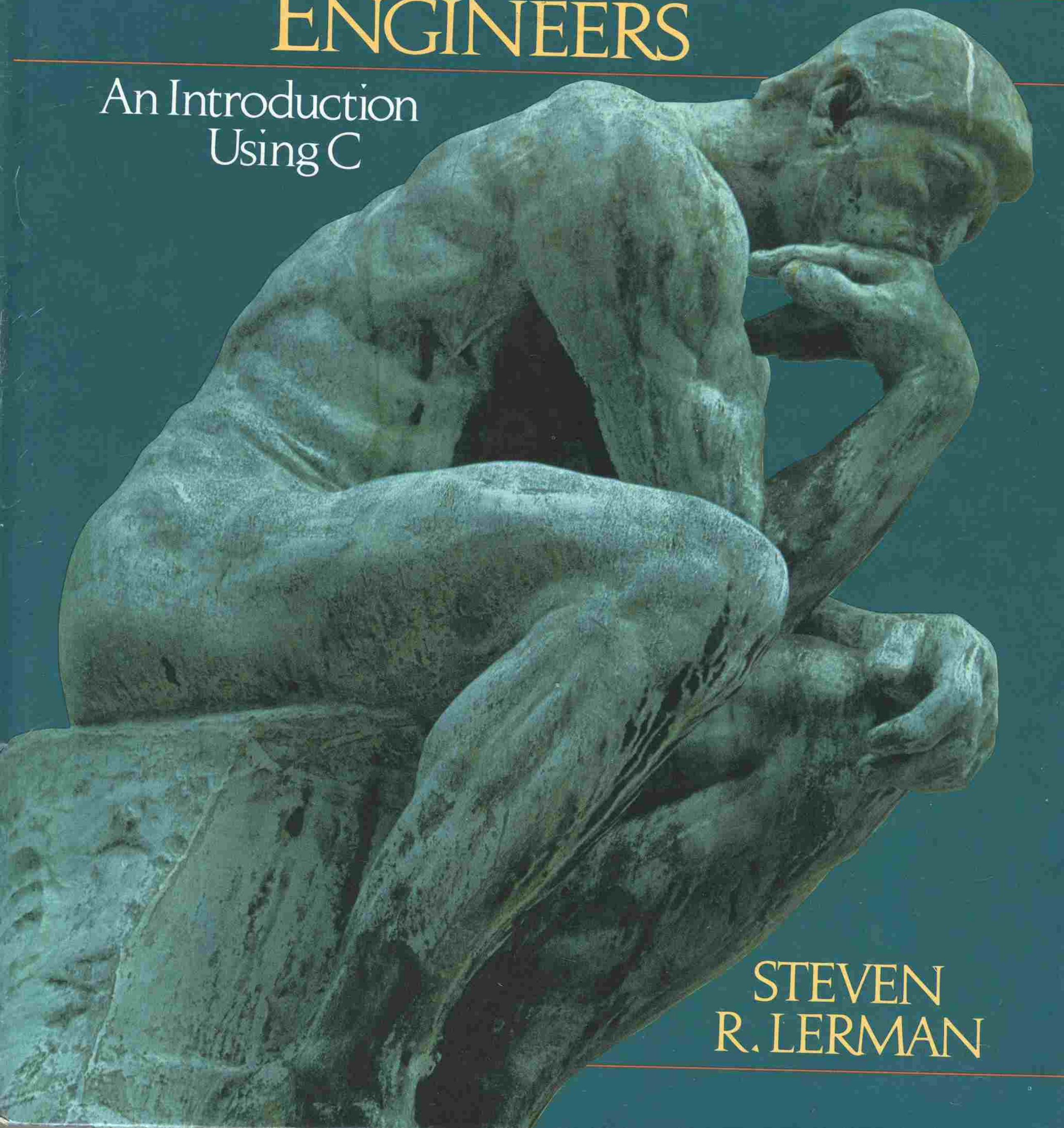
PROBLEM SOLVING AND COMPUTATION FOR SCIENTISTS AND ENGINEERS

An Introduction Using C



STEVEN R. LERMAN

CHAPTER

3

NUMERICAL ANALYSIS

3.1 INTRODUCTION

It is quite common for scientists and engineers to formulate *mathematical models* of the systems they study and design. These mathematical models usually describe complicated phenomena by one or more equations. Unfortunately, many of these systems of equations cannot be solved mathematically. Instead, people who study such systems rely on computers to develop approximate solutions. The algorithms used for this purpose are generally grouped under the term *numerical analysis*.

For example, the paths of celestial bodies moving in space can be theoretically described by the straightforward application of Newton's laws of gravitation. Given the masses of all the bodies that exert any significant force on an object and the objects' current velocities, the movement of each body in space is described by two equations. First, the net force exerted on the body is the sum of the gravitational forces that all the other bodies exert. Each pair of bodies are attracted by a force that is proportional to the product of their masses and inversely proportional to the square of the distance between them. Second, the movement of each body is described by the equation $F = ma$, where F is the net gravitational force acting on each body, m is mass of the body, and a is the body's acceleration.

It would seem that such a simple system of equations would be straightforward to solve. In fact, when there are more than two bodies involved, solution of the equations

of motion is extremely complex. Scientists rely on computational methods to solve these equations in numerical rather than symbolic form.

Another motivation for the development and application of algorithms for numerical analysis is that some equations are demonstrably unsolvable. Consider, for example, the so-called "bell-shaped curve" (or more properly, the normal or Gaussian distribution) that is often used to characterize the distribution of students' performance on a test or an attribute such as intelligence or physical strength. Suppose that one is interested in the area underneath the normal curve shown in Figure 3.1 between two points on the x axis. Even though the equation describing this distribution is relatively simple, it can be shown that there exists no closed-form solution for the area under this curve. Numerical algorithms are needed to solve such problems.

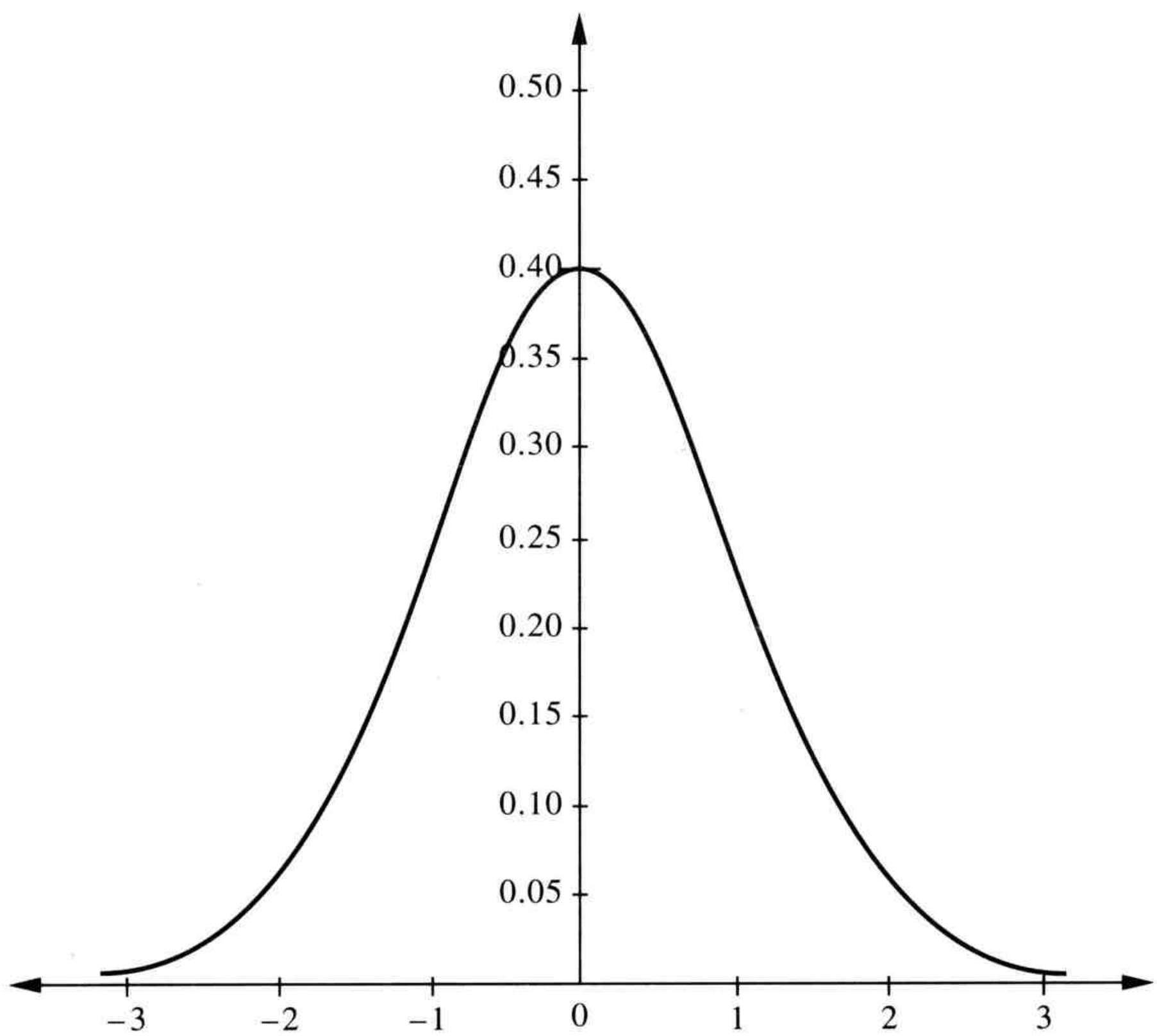


Figure 3.1 Area under the normal distribution curve.

In this chapter we introduce some of the algorithms used in numerical analysis. We try to give the reader a sense about both the value of such methods and the potential hazards of their indiscriminate use. In particular, many standard programs exist for solving numerical problems, and it is often important for the user to understand both the strengths and weaknesses of the procedures these programs use. As we shall discuss further, many numerical analysis methods can give results that are seriously in error, and the causes of the errors may be difficult to detect.

Our study of numerical methods will consider the following topics:

- Representation of floating-point numbers, and the sources of error when various arithmetic operations are performed.
- Solving for the roots of an equation.
- Numerical integration.
- Solution of simple, ordinary differential equations.

3.2 FLOATING-POINT REPRESENTATION

C

In Chapter 2 we introduced the concept of floating-point variables. In C, floating-point numbers are implemented using the types **`float`**, **`double`**, and **`long double`**. We stated that a number in floating-point representation has a sign, a fraction part, and an exponent. The three floating-point types can differ in the range of exponents they can represent and the number of significant digits in their fraction part.

In this section we explore some of the consequences of using floating-point representation. The major issues covered in this section are applicable to all computers. However, the details of floating-point representation do vary, and some of the specific examples in this section may not apply to all computers.

3.2.1 Typical Implementation

Consider a typical case in which a variable of type **`float`** is stored in 32 bits (binary zeros and ones). For the purposes of this chapter, we assume that the 32 bits are used as follows:

- The first bit is used to represent the sign of the number, with a zero indicating a positive value and a one indicating a negative value.
- The next eight bits are used to represent the base 2 exponent. The exponent of any floating-point number is the value of these eight bits (in base 2) minus a fixed number called the *bias*. In our examples we will use a bias of 127. This would often be referred to as *excess* 127 representation.
- The remaining 23 bits are the fraction part. This value is represented in the base 2 equivalent of decimal notation, with the first binary digit representing units of 1, the second representing units of 1/2, the third representing units of 1/4, ..., and the last digit representing units of 2^{-22}.

The format above is typical for floating-point numbers. However, the actual size of a **`float`** as well as the number of digits used for the fraction part and exponent vary across computers. The representations for variables of type **`double`** and **`long double`** use a similar format but generally use more than 32 bits. (The name **`double`** comes from the fact that on many computers a **`double`** occupies 64 bits while a **`float`** occupies 32.) This allows **`double`**s to have greater precision, and in some cases, exponents that are greater in magnitude.

The following are examples of floating-point values in this representation.

```
3   = 0 10000000 11000000000000000000000 = 3/2 × 2^1

-.5 = 1 01111110 10000000000000000000000 =  -1 × 2^-1

1/8 = 0 01111100 10000000000000000000000 =   1 × 2^-3
```

One characteristic of the representation chosen is that there is more than a single way to represent any given value. In particular, as long as the last digit of the fraction part is zero, one can shift the fraction part right one digit (multiplying it by 1/2) and increase the exponent by one (multiplying the entire number by 2) without changing the value.

For example, the following are all equivalent representations of the number 1/16:

```
1/16 = 0 01111110 00010000000000000000000 = 1/8 × 2^-1
       0 01111101 00100000000000000000000 = 1/4 × 2^-2
       0 01111100 01000000000000000000000 = 1/2 × 2^-3
       0 01111011 10000000000000000000000 =   1 × 2^-4
```

We follow the convention that the fraction part of any floating-point number is as "left-shifted" as possible. In this left-shifted form, the first digit of the fraction part, corresponding to the "ones" place, is always one. In the example showing four equivalent representations of 1/16, the last of them would be used. This normalizes the fractional part to be between 1 and 2. Since in this form the first digit of the fraction part is always known, many computer architectures do not even store the first digit of the fraction part, relying on the fact that it is always one. This gives an extra bit of precision in the fraction part. In this form, the representation of 1/16 would be

```
1/16 = 0 01111011 00000000000000000000000
```

The representation described above, including the assumption that the first binary digit of the fractional part is one and that it therefore need not be represented explicitly, corresponds to the IEEE (Institute of Electrical and Electronics Engineers) 754 standard [Institute of Electrical and Electronics Engineers 85] for 32-bit floating-point numbers. This is often called *single precision*, and in many C implementations corresponds to the **`float`** data type. This IEEE standard is used widely (though not universally). Similarly, the same standard specifies that 64-bit floating-point numbers use a sign bit, 11 bits for the exponent (in excess 1023 representation), and the remaining 52 bits for the fractional part, with the first place of the fractional part not represented and always assumed to be one. This is often called *double precision*, and corresponds in many C implementations to the **`double`** type.

The rules for representing floating-point numbers also leave some ambiguity about how to represent the value 0.0. The standard convention is that a floating-point value with all the binary digits set to zero represents the value floating point 0.0.

3.2.2 Overflow and Underflow

In Chapter 2 we stated that there is a largest and smallest value that can be represented by any floating-point number. Using the hypothetical representation of a `float` described above, the largest number would correspond to all bits except the sign set to one. This would be a decimal value that is very close to 2^{128}. The same limitation applies to the absolute value of negative numbers.

An attempt to compute a `float` larger than the greatest representable value produces what is called an *overflow*. What happens in this case is entirely machine dependent. It is generally up to the programmer to detect cases where overflow is likely to occur and take appropriate actions such as warning the user of the program that the results are likely to be highly suspect. In some cases the computer hardware sets a special indicator when a floating-point overflow occurs. The programmer can test when this indicator is set and take appropriate action in the program.

A related possible condition is called *underflow*. This occurs when computations yield values that are too close to zero to be represented. Using our hypothetical floating-point representation, any value closer to zero than 2^{-127} in absolute magnitude is not representable. The C language definition treats the action taken in case of underflow as machine dependent. However, many computers produce a zero as the result of a floating-point computation that underflows.

3.2.3 Limitations on Precision

The fact that the fraction part of a floating-point number has a finite number of bits means that most values cannot be represented exactly. For example, the value 1/3 is not representable exactly in a finite number of decimal digits or in floating-point form. In our example, any quantity with more than 2^{24} binary digits is not exactly representable as a `float`. A similar limit will exist for the other floating-point types.

What is somewhat more subtle is that even if two numbers are both exactly representable as floating-point values, their sum may not be. This can be seen using our hypothetical `float` representation. The values 5 and $(1/2)^{-24}$ can both be represented exactly in our floating-point form. However, their sum cannot be represented. In fact, their sum has the same representation as the number 5! This illustrates the more general problem that if two floating-point numbers to be added differ in magnitude by an amount that is larger than the precision of the floating-point representation for that type of value, then the smaller of them is in effect treated as a zero.

The largest number which can be added to the floating-point representation value for 1.0 without changing the value to something other than 1.0 is called the *machine accuracy*. Almost any arithmetic operation on floating-point values can introduce fractional errors of at least the machine accuracy. This is often called *roundoff error*.

Roundoff errors can be considerably greater than the machine accuracy. The most serious roundoff errors are often introduced when quantities of very different magnitudes are subtracted. Knuth[Knuth, Donald E. 73b] gives the example of a hypothetical computer that works in eight digit decimal notation (rather than binary).

The computation

```
(11111113 - 11111111) + 7.511111
```

on such a computer produces the value 9.5111111, while the mathematically equivalent computation

```
11111113 +  ( 7.511111 - 11111111)
```

produces the result 10.000000.

One of the important consequences of the roundoff error introduced by floating-point arithmetic is that you almost never want to test two computed floating-point values for equality. For example, on many computers the following program will produce a loop that never terminates:

```
/* test for inequality in floating-point arithmetic */

#include <stdio.h>

main()

{
  float a, c;
  c = 0.0;
  a = 0.0;

  while( a !=  10.0) {    /* loops forever */
    c = c+a;
    a = a + .2;
  }
  printf("Value of c = %f\n", c);
}
```

The intent of the programmer was to sum the values between 0 and 10 in increments of .2. However, since the value .2 cannot be represented exactly in a finite number of bits, roundoff errors in incrementing the variable **a** will accumulate, and **a** may never be exactly 10.0. Thus, the inequality test in the **while** loop will never be false, and the program will loop forever.

The appropriate way to code this program is to have the loop continue while **a < 10.00**. More generally, loops that must terminate on some condition that involves comparisons of floating-point numbers should check if the floating-point values are *nearly* identical, that is, whether they are within some reasonably small value of each other. This "reasonably small value" must be chosen to be small enough so that the condition is not incorrectly met and large enough to be meaningful when compared to the machine accuracy.

All of the discussion of floating-point arithmetic above should serve as a warning to anyone intending to do serious numerical analysis on a computer. The issues are not merely ones of technical interest. Failure to account for the fact that floating-point arithmetic always introduces errors can produce results that are completely misleading. The reader should approach any use of floating-point arithmetic with some caution.

3.2.4 Header file float.h

The ANSI C standard specifies that a header file named **`float.h`** must be provided in every implementation. This file defines constants which reflect the characteristics of that machine's floating-point arithmetic. Entries in this file include items such as the largest and smallest representable **`float`** and **`double`** and the number of significant digits in the floating-point types.

The most widely used values defined in **`float.h`** are given in the following table.

SOME ITEMS IN FLOAT.H

Name	Meaning
`FLT_DIG`	number of decimal digits of precision for **`float`**
`DBL_DIG`	number of decimal digits of precision for **`double`**
`FLT_EPSILON`	machine precision for **`float`**
`DBL_EPSILON`	machine precision for **`double`**
`FLT_MAX`	maximum **`float`**
`DBL_MAX`	maximum **`double`**
`FLT_MIN`	minimum positive **`float`** (other than zero)
`DBL_MIN`	minimum positive **`double`** (other than zero)

The ANSI C standard also gives minimum and maximum allowed values for all of the constants above. These values are given in the following table.

REQUIRED VALUES IN FLOAT.H

Name	Requirement
`FLT_DIG`	6 or greater
`DBL_DIG`	10 or greater
`FLT_EPSILON`	10^{-5} or smaller
`DBL_EPSILON`	10^{-9} or smaller
`FLT_MAX`	10^{37} or greater
`DBL_MAX`	10^{37} or greater
`FLT_MIN`	10^{-37} or smaller
`DBL_MIN`	10^{-37} or smaller

The values in **float.h** can be used to test for possible overflow before it occurs. For example, we might want to read floating-point values known to be positive into a **double** and test the magnitude of the **double** before assigning it to a **float**. The following fragment illustrates this:

```
/* fragment using values in float.h */

#include <stdio.h>
#include <float.h>

double temp;
float value;

scanf("%lf", &temp);
if(temp > FLT_MAX || temp < FLT_MIN)
  printf("Unable to assign value to float \n");
else
  value=temp;
```

The reader is referred to [American National Standards Institute 88] for further details on the contents of the header file **float.h**.

3.3 THE STANDARD C MATHEMATICAL FUNCTION LIBRARY

C

In Chapter 2 we introduced the standard C input and output library and noted that almost all C programs use the preprocessor directive

```
#include <stdio.h>
```

which includes the standard header file for this library in the program. C also provides other libraries for the programmer, including one that provides a set of mathematical functions that are commonly used by engineers and scientists. These functions are carefully constructed by professional programmers to be as accurate and efficient as possible. C programs that use these functions should have the preprocessor directive

```
#include <math.h>
```

in the code.

The reader should be aware that unlike the standard C library, the C mathematical functions are usually not automatically linked into a compiled program. The programmer must generally tell the compiler which libraries of functions (other than the standard set) he or she is using. There is no C language standard for how this is done, but in most C compilers the command that invokes the compiler has an option of the form

`-l` (the `l` is a lowercase letter 'el'). This option is followed by an abbreviation for the particular library. For the mathematical function library, the usual command line option is the form `-lm`.

The ANSI standard for the the C language specifies that the functions listed in the following table should be available in the mathematical library. The table lists only a subset of the mathematical functions provided in C. (The complete list is given in the appendix.) All of these functions return a value of type `double`. The returned value is generally the result of evaluating a mathematical function. For example, the C function `cos(x)` returns a `double` which is the cosine of its argument `x`.

MATH FUNCTION LIBRARY

Function	Arguments	Value Returned by Function
`acos()`	`double x`	arc cosine
`asin()`	`double x`	arc sine
`atan()`	`double x`	arc tangent
`atan2()`	`double y, double x`	arc tangent of `y/x`
`ceil()`	`double x`	smallest `double >= x` that can be represented as an `int`
`cos()`	`double x`	cosine
`cosh()`	`double x`	hyperbolic cosine
`exp()`	`double x`	exponential function
`fabs()`	`double x`	absolute value
`floor()`	`double x`	largest `double <= x` that can be represented as an `int`
`ldexp()`	`double x, int n`	computes $x*2^n$
`log()`	`double x`	natural logarithm
`log10()`	`double x`	logarithm base 10
`pow()`	`double x, double y`	x^y
`sin()`	`double x`	sine
`sinh()`	`double x`	hyperbolic sine
`sqrt()`	`double x`	square root
`tan()`	`double x`	tangent
`tanh()`	`double x`	hyperbolic tangent

All of the trigonometric functions in the C math library use radians to measure angles. Similarly, the arc cosine, arc sine, and arc tangent functions all return angles measured in radians. The arc cosine returns a value between 0 and π. The arc sine and arc tangent functions both return values between $-\pi/2$ and $\pi/2$.

The function `atan2()` uses the signs of its two arguments to determine the quadrant for the arc tangent of `y/x`. It returns a value in radians between $-\pi$ and π.

The math library functions can generate two types of errors. For example, the arc sine and arc cosine functions cannot be evaluated meaningfully when their arguments

are greater than 1 or less than -1. Cases such as this where the arguments to a function are inappropriate are said to produce a *domain* error. The other possible error type is where the value to be returned is too large in absolute value to be represented in a **double**. This is called a *range* error.

Errors from library routines are signaled in C through a special variable named **errno**.[1] This **int** is an external, global value which is defined in the header file **errno.h**. Library routines set the value of **errno** to one of a list of predefined error codes whenever an error is found. Each error code is some positive value.

For example, the following fragment of C code uses the error detection capability to determine whether the result of computing x^y produced an overflow error.

```
/* fragment of code showing detection of errors
      in math library functions */

#include <stdio.h>
#include <math.h>
#include <errno.h>

double x,y,z;

errno = 0;  /* set value of errno */

x = 5.0;    /* set values of x and y */
y = 100.0;

z= pow(x,y);  /* compute x to the y power */

if(errno != 0)  /* check if error found */
  printf("Error in computing %lf to the %lf\n",
                x, y);
```

In this code fragment, the three **#include** statements include the standard header file, the math library header file, and the error header file. The math library function **pow()** for computing x^y is invoked and the value of the variable **errno** is checked to determine if an error was encountered. The reader should be aware that the value of **errno** is not automatically reset to zero when a library routine is called. It is up to the programmer to reset **errno**.

In order to make a determination of the source of an error easier, the header file **errno.h** has a series of **#define** statements that give names to various integer values. The two names **EDOM** and **ERANGE** correspond to domain and range errors.

[1]Technically, the value **errno** may not actually be a C variable. It may be the result of a **#define** preprocessor directive which expands into an integer value which can be used on the left-hand side of an assignment statement.

Thus, the fragment above could be rewritten to be clearer about the nature of the error encountered as follows:

```
/* fragment of code showing detection of errors
      in math library functions */

#include <stdio.h>
#include <math.h>
#include <errno.h>

double x,y,z;

errno = 0;  /* set value of errno */

x = 5.0;    /* set values of x and y */

y = 100.0;

z= pow(x,y);  /* compute x to the y power */

if(errno == EDOM)  /* check  for domain error */
  printf("Domain error in computing %lf to the %lf\n",
                x, y);

else if(errno == ERANGE) /* check for range error */
  printf("Range error in computing %lf to the %lf\n",
                x, y);
```

3.4 POINTERS TO FUNCTIONS

C

All of the algorithms we present in this section have as one of their arguments a function. C provides a mechanism for using functions as arguments. The central idea behind this concept is that a *pointer to a function* is a valid variable type.

At first, the idea of passing a function as an argument may seem strange. However, imagine that we wanted to code a C function that computed a sum of terms, where each term was the value of a function evaluated at a different point. If we had to compute many such sums, it would be handy to have a single function that summed *any* set of terms. We could then pass the function that computes the specific terms of any series to the more general summation function as an argument.

For example, suppose that we wanted to compute the values of the following three series:

$$x^{.1} + x^{1.1} + x^{2.1} + x^{3.1} + \cdots + x^{14.1}$$

$$x^{1.5} + x^{2.5} + x^{3.5} + \cdots + x^{12.5}$$

$$x^{5.25} + x^{6.25} + x^{7.25} + \cdots + x^{9.25}$$

One could code a function for each of these summations, but this would require a new function for every new series encountered. A better alternative would be to code a general-purpose program that computed the value of any series of the form

$$\sum_{i=a}^{b} f(x, i)$$

Once such a function was written, you could write a small function for each $f(x, i)$ and pass this new function to the more general procedure as an argument. In the three series given above, the terms have the forms $x^{i+0.1}$, $x^{i+0.5}$, and $x^{i+0.25}$, respectively.

Several examples will serve to demonstrate how pointers to functions are declared as arguments. Consider, for example, an argument **f** which is a function that has a **double** and an **int** as arguments and returns a **double**. This function might evaluate a single term in a series such as $x^{i+0.1}$. It would have two arguments: the values of x (in a **double**) and i (in an **int**). If this function is an argument to some other function, it would be declared in that other function's prototype as follows:

```
double (*f) (double, int)
```

Note the use of the parentheses in **(*f)()**. This is because the declaration

```
double *f(double, int)
```

declares **f()** to be a function that returns a *pointer to a* **double** rather than a pointer to a function that returns a **double**. The parentheses denoting a function have higher precedence than the *. The use of the first pair of the parentheses in **(*f)()** overrides the normal order of precedence.

With this in mind, we can write our general-purpose routine that sums a series of terms as follows:

```
/* general-purpose summation routine */

double summation(
   double x,                        /* value of x */
   double (*f)(double, int),        /* function for individual
                                                        terms */
   int a,                           /* initial index for sum */
   int b)                           /* final index for sum */
{
  double sum;

  for(sum = 0.0; a <= b; a=a+1)
    sum = sum + f(x, a);
  return(sum);
}
```

In this function, the argument **f** is used just as any other function. Any C variable which is a pointer to a function can be used to invoke the function it points to. In fact, the name of a function can be thought of as a pointer to that function. When we want to invoke that function, we follow the name by an argument list in parentheses.

The function prototype for **summation()** would be as follows:

```
double summation(double,double (*)(double,int),int,int);
```

We would use the function **summation()** as follows:

```
\* example of program using summation */

#include <stdio.h> /* I/O header file */
#include <math.h>  /* math header file */
double summation(double,double (*)(double,int),int,int);
double f1(double,int), f2(double,int), f3(double,int);

main()
{
  double total,x;

  printf("Enter value of x for series:");
  scanf("%lf", &x);

  total = summation(x, f1, 0, 14);
  printf("First series sum is %lf\n", total);

  total = summation(x, f2, 1, 12);
  printf("Second series sum is %lf\n", total);

  total = summation(x, f3, 5, 9);
  printf("Third series sum is %lf\n", total);

}


/* terms for first series */

double f1(double x, int i)
{
  return( pow(x, i+.1));
}


/* terms for second series */

double f2(double x, int i)
{
```

```
  return( pow(x, i+.5));
}


/* terms for third series */

double f3(double x, int i)
{
  return( pow(x, i+.25));
}
```

This example uses the standard C mathematical library function **pow()**. This function has two **double**s as its arguments. **pow(x,y)** computes x^y.

The ability to pass pointers to functions makes it possible to code general-purpose libraries of functions that solve a range of problems. This avoids the need to recode complicated procedures to customize them to work with different functions. We will see the usefulness of this in the following sections.

3.5 ROOT FINDING

One common problem encountered in many forms of mathematical analysis is finding the roots of some equation. For example, consider the equation

$$x^4 - 8x^3 + 22x^2 - 24x + 9 = 0$$

The values of x which satisfy this equation are called its *roots*. This equation can have as many as four real roots. While some equations (such as this one) can be solved analytically, many important ones can only be solved using numerical methods.

Root-finding problems arise in many contexts. For example, in predicting how many people might use a particular highway, transportation analysts often assume that there is both a demand for travel and an ability of the highway facility to supply transportation services.

The demand for travel depends on the average travel time. This demand might be represented by the function $V = D(t)$, where t is the travel time on the facility and V is the number of users. As the time required to travel on the highway increases, the demand for it decreases.

The travel time on the facility in turn depends on the demand for its use. As more people try to use the highway, congestion builds up, increasing the travel time. This relationship can be represented by the function $t = S(V)$.

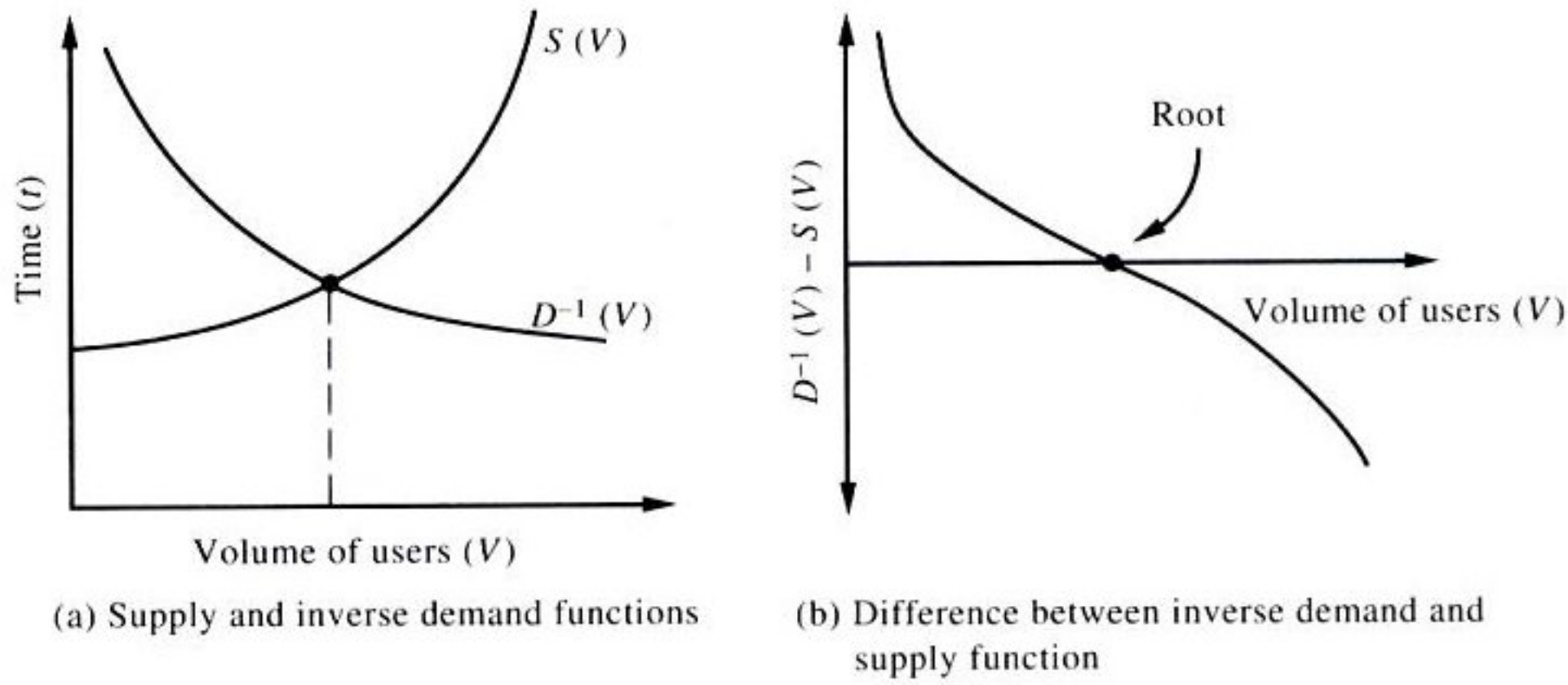
The problem of finding the travel time and volume of users on the highway can be reduced to a root-finding problem by noting that the demand relationship can usually be inverted, expressing time as a function of volume. We denote this as $t = D^{-1}(V)$.[2]

[2]The notation $D^{-1}(V)$ should not be confused with $1/D(V)$. It represents the transformation of the function from V as a function of t to an equivalent form where t is a function of V.

The value of V which satisfies both the inverse of the demand function and $S(V)$ is the root of the equation

$$D^{-1}(V) - S(V) = 0$$

Figure 3.2a and b illustrate the two functions and their difference. The root of $D^{-1}(V) - S(V)$ is also shown.



(a) Supply and inverse demand functions

(b) Difference between inverse demand and supply function

Figure 3.2

This example is a specific case of the more general problem of solving for a point where two functions intersect. If $f_1(x)$ and $f_2(x)$ are two functions, then the root of $f_1(x) - f_2(x)$ will be a value of x at which the functions intersect.

Some of the most useful root-finding methods are described in this section. We restrict our attention to solving for real roots rather than complex roots. Also, while all of the methods we discuss generalize to multivariate functions, our presentation is limited to single-variable equations.

Root-finding methods all share some common characteristics. In particular, the methods we will examine share the following:

1. They all require either a known interval $[x_1, x_2]$ in which a root is to be searched for or some initial starting value of x_0 from which a search is to begin.
2. None of the methods are guaranteed to find *all* the real roots of an equation. Instead, they find at most one root.
3. If they succeed in finding a root, they do so only approximately. The approximate nature of the solution is only in part because they rely on floating-point arithmetic. Another reason is that the methods are intrinsically iterative, converging upon a solution step by step. Even on a computer with infinite precision, the methods would not reach the true root in a finite number of steps.

Despite these limitations, numerical root-finding methods are often the only feasible means of solving many real-world problems.

3.5.1 Finding an Interval Containing a Root

Many of the algorithms that find a root of an equation require as one of their arguments an interval that contains a root. A simple algorithm for finding an interval that contains a root of a function $f(x)$ exploits the fact that any interval $[x_1, x_2]$ where

$$f(x_1) \cdot f(x_2) \leq 0$$

must contain at least one root. In fact, such an interval must contain an odd number of roots. To find such an interval, we take a broad range of x, divide it into subranges, and search for an interval that satisfies the condition above. A C function named **find_interval()** that does this search requires the following arguments:

f — a pointer to the function for which an interval containing a root is to be found.

xstart and **xend** — the starting and ending values of x defining the interval over which a search for an interval containing a root is to be done.

stepsize — the size of the intervals to be tested.

xleft and **xright** — pointers to variables of type **double** that point to the beginning and end of an interval containing a root.

The values **xleft** and **xright** must be pointers because **find_interval()** must be able to change the values of the interval in the function that calls it. If the arguments **xleft** and **xright** were simply **double**s, then **find_interval()** would only be able to change copies of these arguments.

The function **find_interval()** returns an integer which is set to 1 if an interval that contains a root was found, and zero otherwise. The values 1 and 0 are chosen because they can be used as indicators in a logical expression. (Recall that logical expressions in C evaluate to 0 if they are false and to 1 if they are true.)

The implementation of **find_interval()** is as follows:

```
/* function to find an interval containing a root */

#define TRUE 1
#define FALSE 0

int find_interval(
  double (*f)(double), /* function */
  double xstart,       /* start of search */
  double xend,         /* end of search */
  double stepsize,     /* interval of search */
  double  *xleft,      /* pointer to  left
                          of found interval */
  double *xright)      /* pointer to right
                          of found interval */
```

```
{
  int found_root = TRUE; /* initialize indicator */

  /* initialize interval */
  *xleft = xstart;
  *xright = *xleft + stepsize;

  /* search interval */
  while (*xleft < xend && f(*xleft) * f(*xright)>0.0) {
        *xleft = *xright;
        *xright += stepsize;
        if(*xright > xend)  *xright = xend;
     }

  /* check if root found */
  if(f(*xleft) * f(*xright)> 0.0)
        found_root = FALSE;

  return(found_root);
}
```

This implementation of **find_interval()** assumes that the value of **stepsize** is less than or equal to **xend-xstart**. It is coded to avoid attempting to evaluate the function **f()** outside the range of the initial interval. This is to avoid the possibility that the function is not defined for values of x smaller than **xstart** or larger than **xend**.

A typical use of **find_interval()** to find an interval between -10 and 10 in which the function

$$f(x) = x^3 - 4x^2 - 4x + 15$$

contains a root would be as follows:

```
/* program that uses find_interval() */

#include <stdio.h>
main()
{
  double polycubic(double);
  int find_interval(double (*f)(double), double, double,
                          double, double *, double *);
  double x1, x2;

  if(find_interval(polycubic, -10.0, 10.0,
                   0.1, &x1, &x2))
       printf("Interval is %lf to %lf\n", x1, x2);
  else
       printf("No interval was found\n");
}
```

```
/* function whose root to find */
double polycubic(double x)
{
  return(x*x*x - 4*x*x - 4*x + 15);
}
```

Note that the algorithm above finds only the first interval containing the root and may fail to find an interval if an even number of roots are sufficiently close to one another. The first shortcoming can be alleviated by applying the function **find_interval()** again after a first interval is found, searching from the end of the first interval. The second shortcoming is intrinsic to the algorithm and cannot be eliminated entirely. Plotting the function and knowing its mathematical properties can often provide clues to intervals where roots must lie.

3.5.2 Bisection Method

Assuming that an interval containing a root has been found, the most straightforward approach to root finding is called *bisection*. This method searches over a given interval known to contain at least one root and successively halves that interval, restricting the search to the half known to contain the root. This successive halving converges to a very small interval in which a root must lie. The algorithm terminates when the interval is "small enough" by some criterion. What constitutes an interval which is small enough should be controlled by the user of the algorithm. However, it makes no sense to attempt to use bisection to obtain an interval which, as a fraction of the initial interval, is smaller than the machine accuracy.

Bisection is a classic example of a computational strategy often referred to as *divide and conquer*. The initial interval $[a, b]$ is successively halved, implying that at the nth iteration the size of the interval is $(b - a)/2^n$. Such algorithms converge very quickly to intervals that are small. If the floating-point representation of a **double** has d significant digits in binary form, then bisection will never require more than d iterations to converge to an interval that, as a fraction of $(b - a)$, is as small as the machine accuracy.

Bisection can be implemented either recursively or iteratively. The iterative implementation is as follows:

```
/* bisection method */
#include <math.h>
double bisect(
  double (*f)(double), /* function */
  double x1, /* start of interval */
  double x2, /* end of interval */
  double epsilon) /* convergence tolerance */
{
  double y;
  for(y=(x1+x2)/2.0; fabs(x1-y)>epsilon; y=(x1+x2)/2.0)
```

```
    if(f(x1)*f(y)<= 0.0)
        x2 = y; /* use left subinterval */
    else
        x1 = y; /* use right subinterval */
  return(y);
}
```

The function **fabs()** is a standard C mathematical library function. It takes a single argument of type **double** and returns a **double** which is the absolute value of the original argument. The prototype for **fabs()** is part of the header file **math.h**.

The recursive implementation of bisection is as follows:

```
/* recursive version of bisection */
#include <math.h>

double rbisect(
  double (*f)(double), /* function */
  double x1,           /* start of interval */
  double x2,           /* end of interval */
  double epsilon)      /* tolerance */
{
  double y;

  y = (x1+x2)/2.0;
  if( fabs(x1-y) > epsilon) /* check for convergence */
    if(f(x1)*f(y) <=0)
      y = rbisect(f,x1,y,epsilon);/* use left side */
    else
      y = rbisect(f,y,x2,epsilon);/* use right side */
  return(y);
}
```

In actual practice, the iterative version would be used because it is somewhat more efficient and probably clearer to most users.

3.5.3 Secant Method

The *secant method* uses an entirely different approach to root finding than the bisection method. The algorithm works by fitting a straight line between the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. It then finds x^*, the root of the straight line. If the result is sufficiently close to x_2, it returns that value as the root of the original equation. If not, it resets the value of x_1 to x_2 and x_2 to x^*, and tries the method again. This proceeds until a solution close to x_2 is found or some number of attempts to find a solution have failed.

Figure 3.3 shows a typical sequence of steps of the secant method. The points (x_1, y_1) and (x_2, y_2) are the starting points for the algorithm. The line drawn between these two points is the secant, and the point $(x_3, 0)$ is where the secant intersects the x

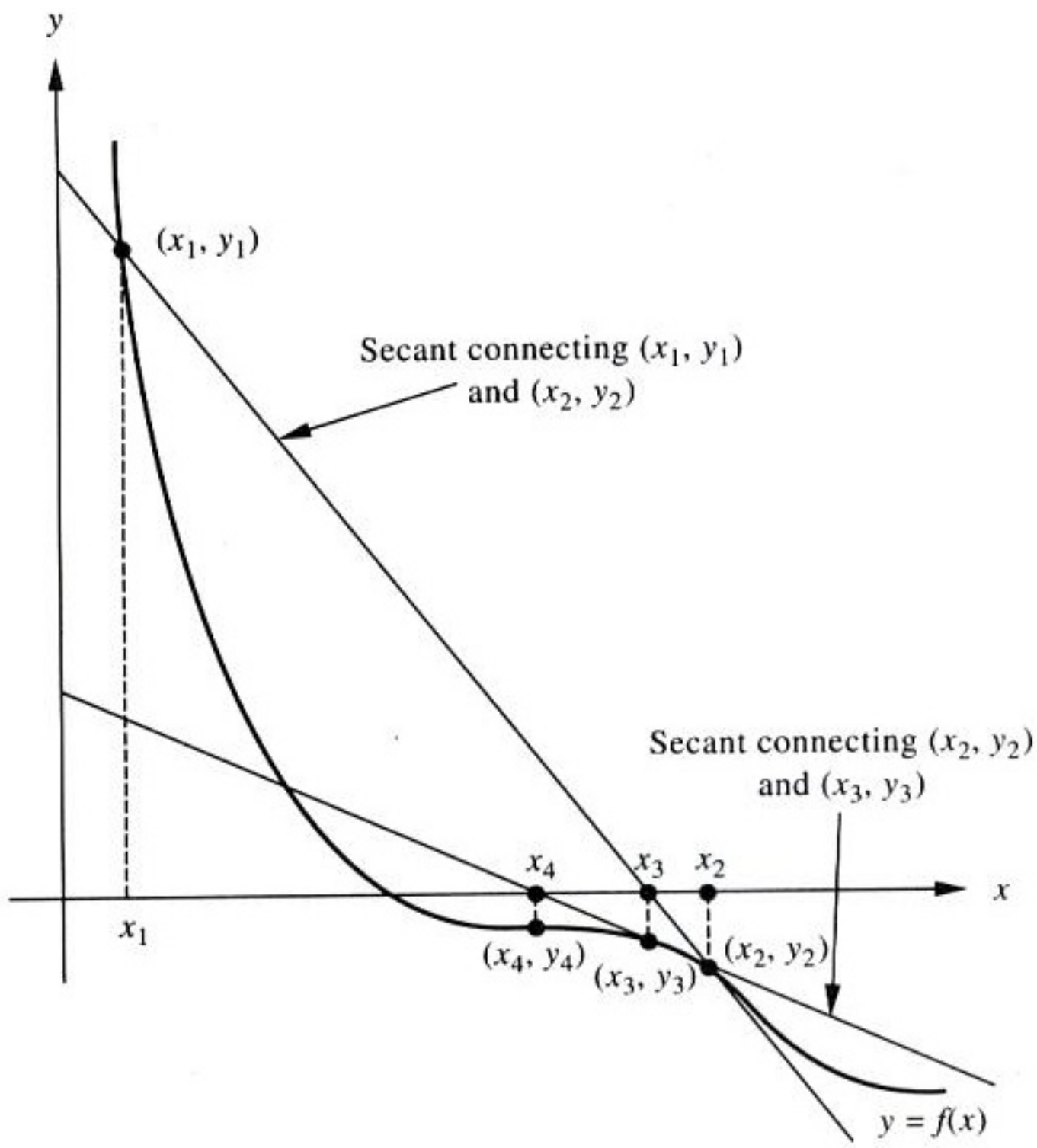


Figure 3.3 Secant method.

axis. The next step in the algorithm would use the line between (x_2, y_2) and (x_3, y_3) to define the next secant, producing the value x_4 as the next estimate of the root. The algorithm proceeds in this fashion until either the convergence criterion is satisfied or some upper limit on the number of iterations is reached.

Implemented in C, the algorithm is as follows:

```
/* implementation of the secant method */
#include <math.h>
#define TRUE 1
#define FALSE 0


double secant(
  double (*f)(double),   /* function to find root of */
  double x1,             /* start of interval */
  double x2,             /* end of interval */
  double epsilon,        /* tolerance */
  int max_tries,         /* maximum number of tries */
  int *found_flag)       /* pointer to flag indicating
                                                 success */
{
```

```
    int count = 0;          /* counter of loops */
    double root = x1;       /* root value */
    *found_flag = TRUE;     /* initialize flag to success */

    while(fabs(x2-x1)>epsilon && count<max_tries) {
      root = x1 - f(x1)*(x2-x1)/(f(x2)-f(x1));
      x1=x2;
      x2 = root;
      count = count + 1;
    }

    if(fabs(x2-x1) > epsilon)/* check for convergence */
      *found_flag = FALSE;

    return(root);           /* return root */

  }
```

In this implementation, the value **found_flag** is a pointer to an integer which is set to **TRUE** (defined as 1) if the algorithm succeeds in finding a root and **FALSE** (defined as 0) if the maximum number of tries is reached without finding a root.

There are a number of shortcomings of the secant method. First, any iteration of the algorithm can result in a value that lies outside the initial interval. This is not true for the bisection method. In addition, there is no way for the algorithm to detect that it will fail to find a root. The implementation above uses a counter for the maximum number of allowed iterations to prevent the function from looping forever. This, however, is an entirely *ad hoc* way of dealing with the algorithm's shortcomings. Finally, the algorithm may encounter significant problems with overflow when the magnitude of **f(x2) - f(x1)** gets very small. This is because the solution for the root of the secant requires division of **f(x1) * (x2 -x1)** by **f(x2) - f(x1)**.

For all these reasons, the secant method is not widely used in practice.

3.5.4 Newton's Method

Unlike bisection and the secant method, Newton's method does not require an initial interval. Instead, it begins the search for a root with some initial guess. It then fits a tangent to $f(x)$ through the guess and finds the root of the tangent. This then becomes the next guess. This process continues until the difference between the guess and the root of the tangent line is small.

Newton's method uses the derivative of $f(x)$ to find the slope of the tangent line. Thus, it requires two functions as inputs: the original function $f(x)$ and the derivative of $f(x)$, which we will denote as $f'(x)$. Each iteration of Newton's method is simply

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Figure 3.4 illustrates two steps in Newton's method. The value x_0 is the initial value for x. The line tangent to $f(x)$ through the point (x_0, y_0) is used to find the value x_1 where the tangent crosses the x axis. This value is then used to fit the tangent to $f(x)$ through the point (x_1, y_1) to find the point x_2. The algorithm continues in this fashion until it converges to the root of $f(x)$.

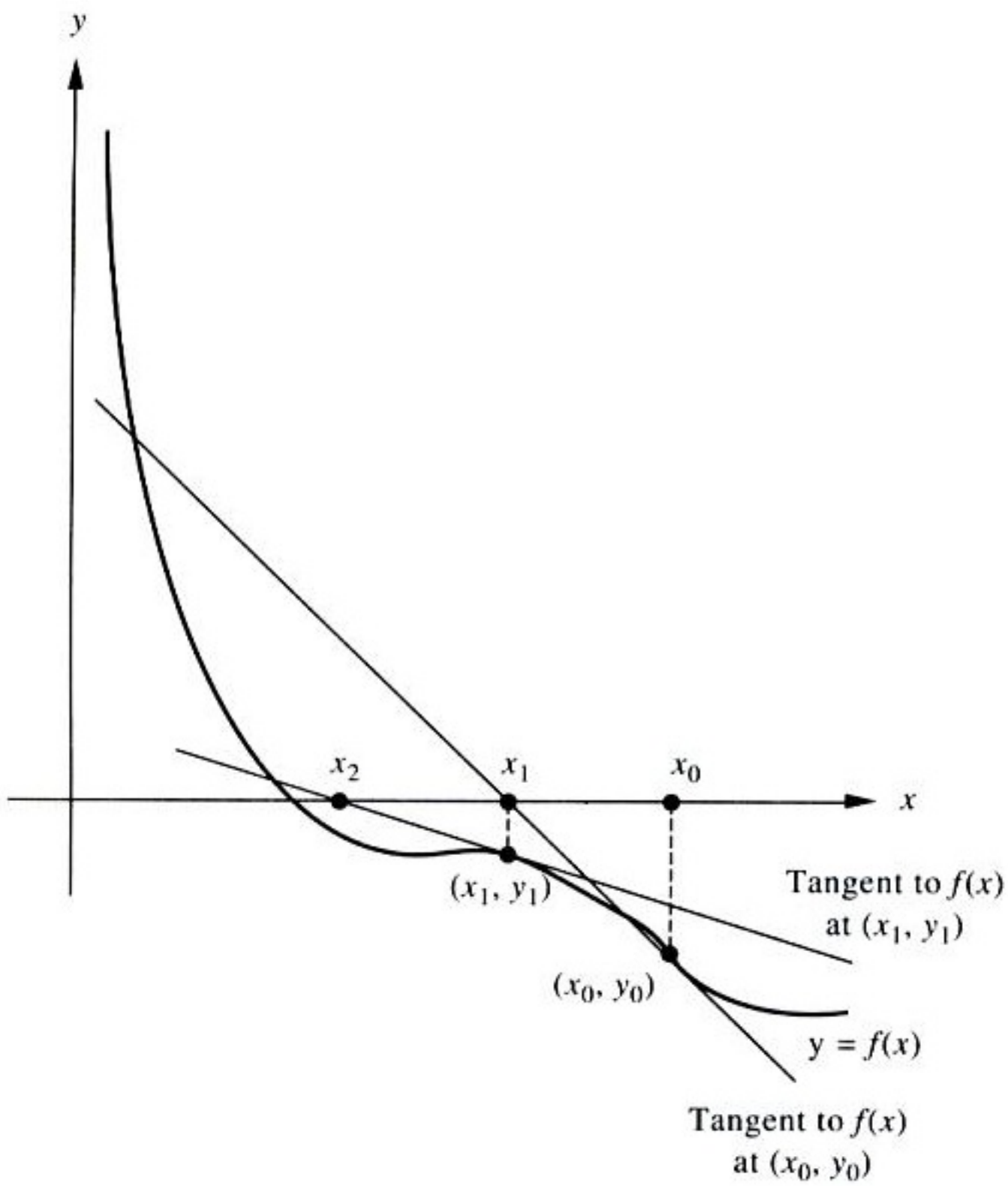


Figure 3.4 Newton's method.

A careful implementation of Newton's method must deal with the possibility that $f'(x_n)$ will be so small that the ratio $f(x_n)/f'(x_n)$ will result in a floating-point overflow. When this occurs, execution should not continue; it should terminate and signal that an error has occurred. In the following C function, the argument **dmin** is the minimum value of $f'(x)$ that will be allowed and the pointer argument **error** is used to change an error indicator in the calling function. The C implementation is as follows:

```
/* implementation of Newton's method */

#include <math.h>
#define TRUE 1
#define FALSE 0

double newton(
  double (*f)(double),      /* function */
```

```
    double (*fprime)(double),/* derivative of f */
    double dmin,              /* minimum allowed value of
                                                  fprime */
    double x0,           /* initial guess of solution */
    double epsilon,      /* convergence tolerance */
    int *error)          /* pointer to error indicator */
  {
    double deltax;
    deltax = 2.0 * epsilon; /*initialize deltax>epsilon */
    *error = FALSE;        /* initialize error indicator */

    while( !(*error) && fabs(deltax) > epsilon)
      if(fabs(fprime(x0)) >dmin)  {
        deltax = f(x0)/fprime(x0);
        x0 = x0 - deltax;
    }
      else
        *error = TRUE;

    return(x0);
  }
```

The function **newton()** above has pointers to functions as its first two arguments. They are used for the functions $f(x)$ and $f'(x)$. The remaining arguments are the minimum allowed derivative value, the starting value x_o, and the convergence tolerance and a pointer to an error indicator. A typical use of **newton()** would be as follows:

```
/* program that uses Newton's method  */
#include <stdio.h>

double polycubic(double);
double  dpolycubic(double);
double newton (double (*)(double), double (*)(double),
                double, double, double, int * );

main()
{
  double root;
  int flag;

  root = newton(polycubic,dpolycubic,10e-30,-1.9,
                0.00001,&flag);
  if(flag)
       printf("Error found in Newton's method\n");
  else
       printf("Value of root is %lf\n", root);
}
```

```
/* function f(x) */
double polycubic(double x)
{
  return(x*x*x - 4*x*x - 4*x + 15);
}

/* derivative of f(x) */
double dpolycubic(double x)
{
  return(3*x*x - 8*x - 4);
}
```

For many problems, Newton's method converges to a root quite quickly. When the method fails to find a root it is often because $f'(x)$ is too small. In such cases it is often possible to select a different initial starting value for which the method will converge to a solution. Newton's method also has the advantage of not requiring an interval containing a root as an input.

3.5.5 Problems In Applying Root Finding Methods

In applying root-finding methods, it is not unusual to encounter a variety of problems reaching convergence on a true root. These difficulties arise from many sources, including the problems of overflow, underflow, and loss of precision in floating-point operations. Such problems are often eliminated simply by selecting a different starting value of x or a different starting interval. However, there are other potential pitfalls in root finding the reader should be aware of.

One problem with the secant and Newton's method as we have presented them is the possibility of *false convergence*. The measure of convergence we have used to test whether the algorithms have reached a root examines the change in the x value between successive iterations. It is possible for this change to be very small even though the value of x is far from a root of the function. Figure 3.5 illustrates how this might occur graphically in the secant method. The points on the x axis labeled x_{i-1} and x_i are the starting values for the secant algorithm. When these two points are used in an iteration of the secant method, they produce the point labeled x_{i+1} as the next estimate of the root. The difference between the endpoint of the initial interval, x_i, and the next estimate of the root, x_{i+1}, is very small in this figure, so the algorithm as coded could terminate. However, when the function $f(x)$ is evaluated, it is clear that neither x_i nor x_{i+1} is even close to a root of the equation.

The solution to this potential problem is that after root-finding algorithms converge in the sense of producing very small changes in the value of x between successive iterations, you should conduct a second check to determine whether the value returned by the algorithm is in fact a root. This check is straightforward; you evaluate $f(x)$ at the value returned and see if by some measure, $f(x)$ is "close to" zero. The exact test as to what constitutes a value of $f(x)$ close to zero depends on the problem at hand.

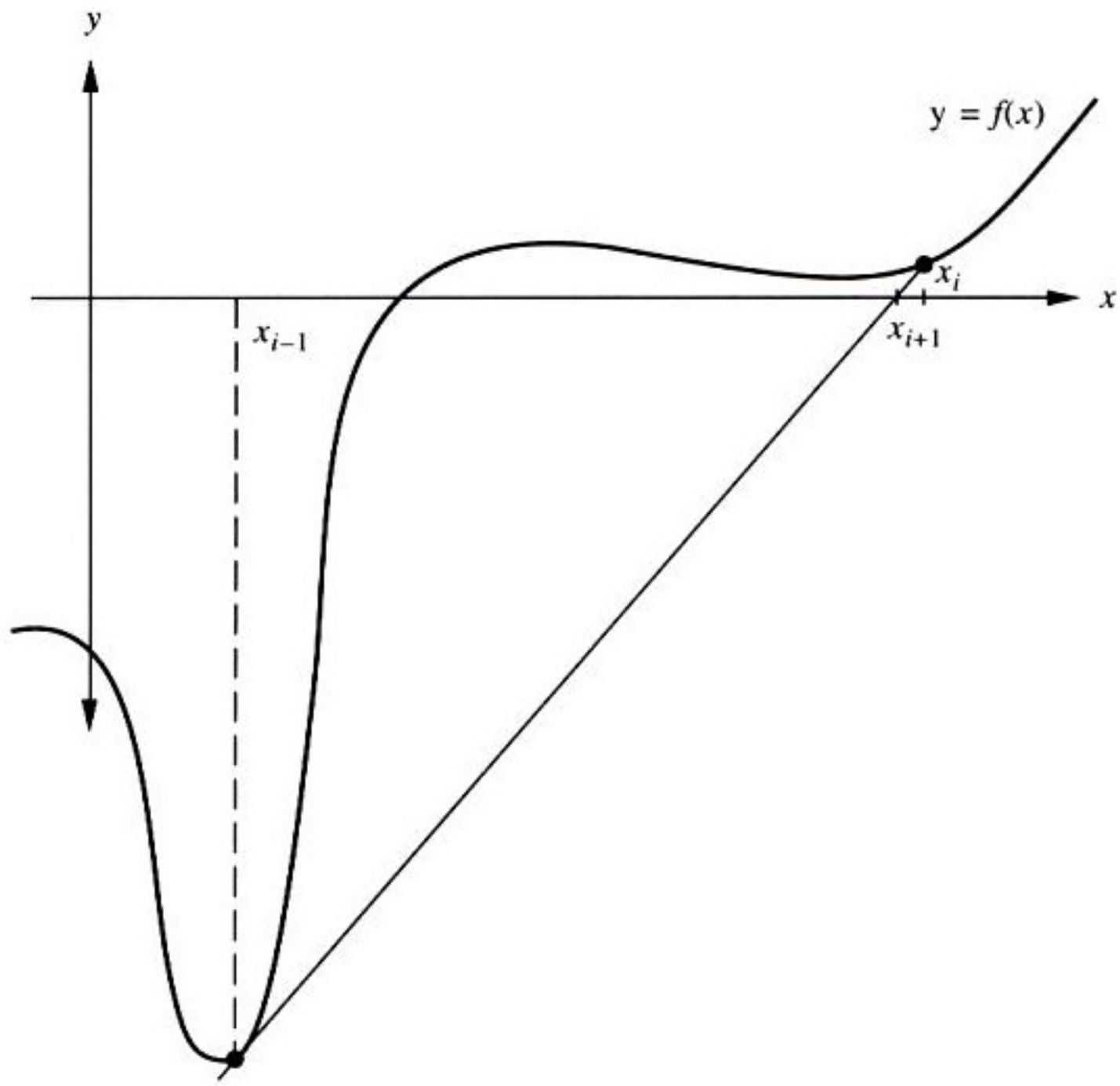


Figure 3.5 False convergence in secant method.

A second potential problem is called *cycling*. This occurs when a series of iterations results in a pattern of values of x which repeats itself. Consider as an example the function

$$f(x) = \frac{1}{1+e^x} - \frac{1}{2}$$

shown in Figure 3.6.

One way Newton's method can cycle when finding the root of this function is for it to oscillate between two values of x on either side of the y axis. For the function shown in Figure 3.6, this type of oscillation will occur for two particular starting values of x_0. The specific values that produce this oscillation are those for which the change in x in one iteration of Newton's method generates a value x_1 exactly equal to $-x_0$. The symmetry of the function around the y axis will guarantee that if x_1 is $-x_0$, then the next value, x_2, will be exactly equal to x_0, producing a complete cycle.

Figure 3.7 shows graphically how Newton's method can cycle. If the point on the x axis labeled x_n is a trial point, then the next iteration of Newton's method will fit a tangent to the function through the point $(x_n, f(x_n))$, producing the point labeled x_{n+1} as the next estimate of the root. The tangent to the function through the point $(x_{n+1}, f(x_{n+1}))$ in turn produces the next estimate of the root as x_n, completing the cycle.

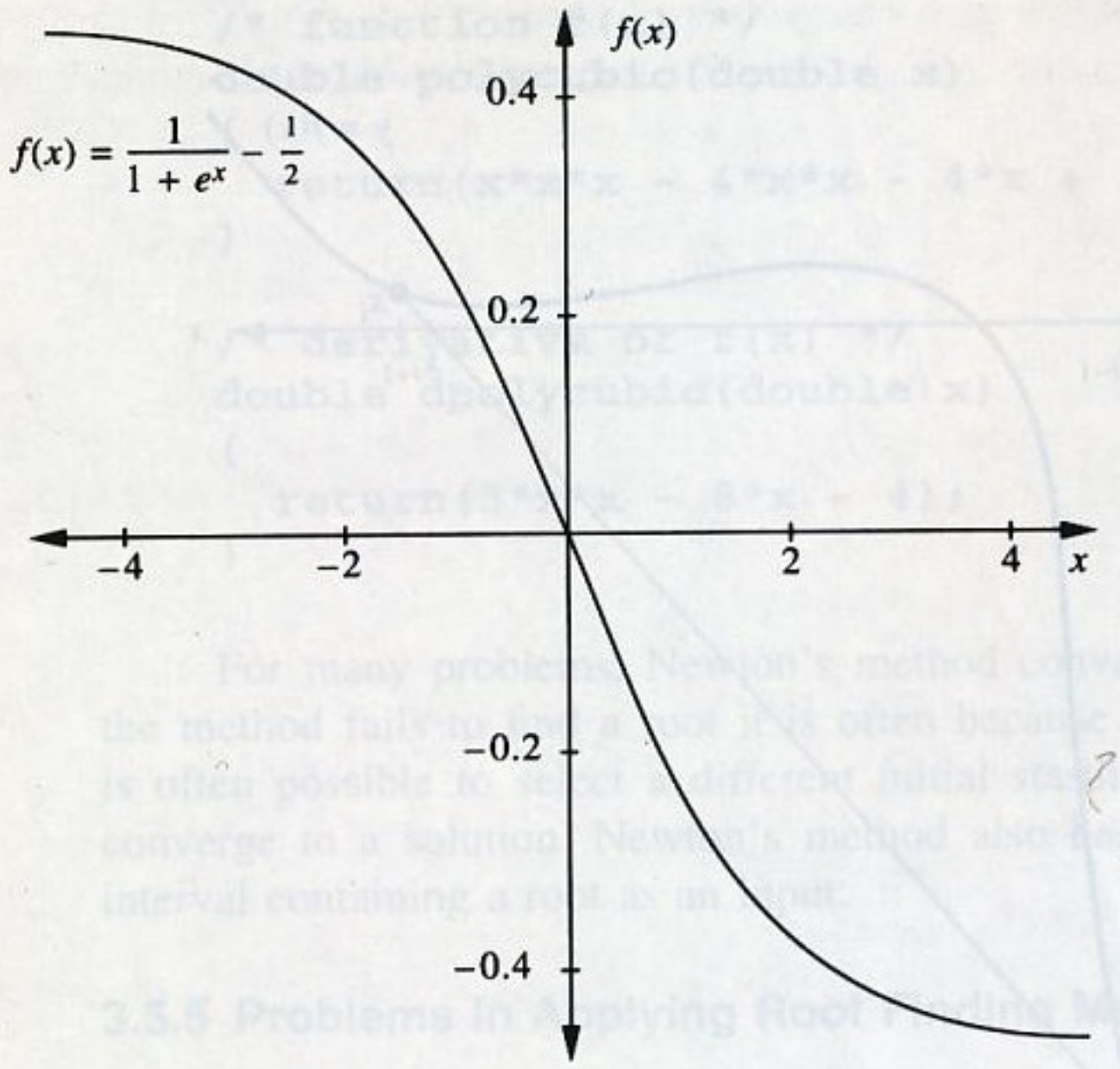


Figure 3.6 Example function for cycling.

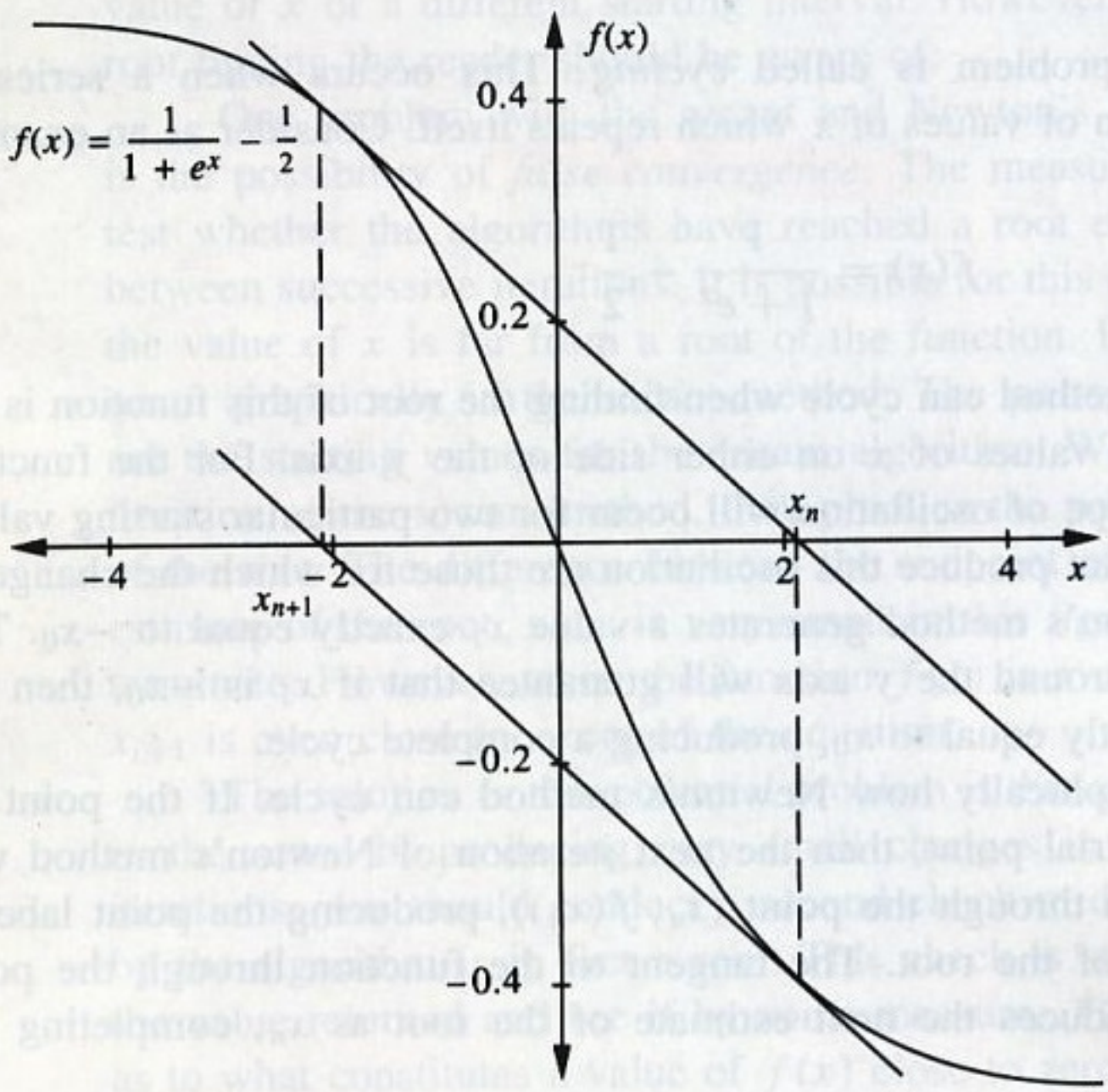


Figure 3.7 Cycling in Newton's method.

We can find the points that produce cycling of Newton's method for a symmetric function such as shown in Figure 3.6 by noting that each iteration of Newton's method is found with the equation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

A value of x such that

$$x_n = -x_{n+1}$$

will satisfy the equation

$$-x_n = x_n - \frac{f(x_n)}{f'(x_n)}$$

or, letting $x = x_n$,

$$2x - \frac{f(x)}{f'(x)} = 0$$

For the equation depicted in Figure 3.6, the reader can verify that x must satisfy

$$2x - \frac{\dfrac{1}{1+e^x} - \dfrac{1}{2}}{\dfrac{-e^x}{(1+e^x)^2}}$$

While there is no analytic solution for the root of this equation, the values of x which solves the foregoing equation are approximately ± 2.177319.

Because of roundoff errors, actual cases of cycling generally take the form of very slow convergence of Newton's method. This can often be alleviated by selecting a different starting value for the algorithm.

3.6 PROGRAMMING PROJECT: ACCELERATION OF A TRAIN

As an example of an application of root-finding methods, we consider a rail rapid transit train which is operating under automatic control. The train is assumed to have a speed governor which limits it to a top speed of 85 kilometers per hour. We will use the bisection method to determine to the closest second how long after the train leaves the station it will attain its top speed.

The train leaves a station at time $t = 0$ seconds and is assumed to accelerate to some maximum velocity. The velocity of the train, $v(t)$, is assumed to be given by the following equation:

$$v(t) = 29.4 + .795t + \frac{1}{(1 - 1.034e^{0.0061667t})}$$

Figure 3.8 depicts the velocity profile determined by the equation above. The x axis is in seconds and the y axis is in kilometers per hour. It is apparent from the figure that the speed limit of 85 kilometers per hour will be attained before 100 seconds has elapsed.

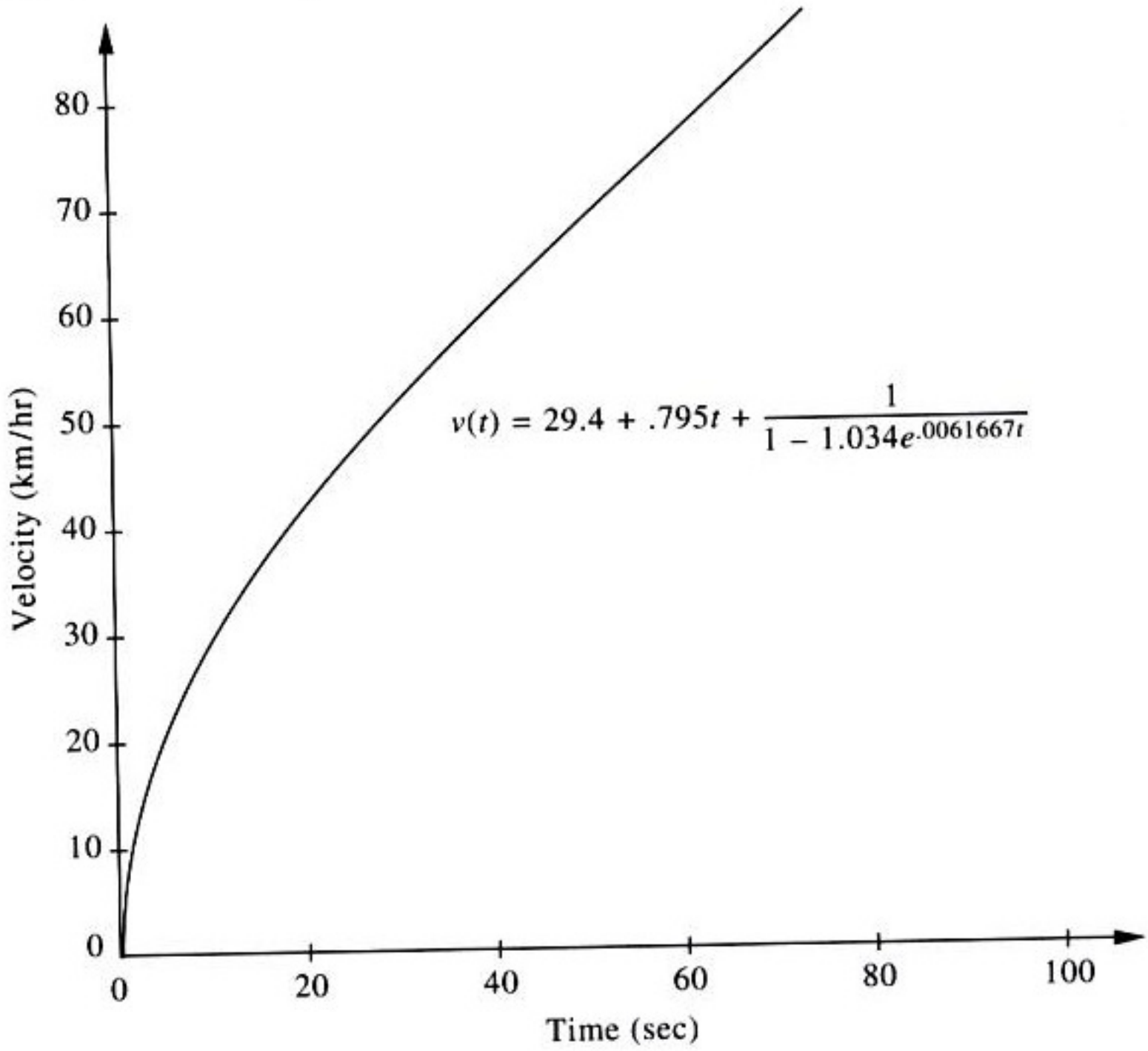


Figure 3.8 Velocity of train vs. time in seconds.

The program that solves this problem begins with the following preprocessor directives and function prototypes:

```
#include <stdio.h>
#include <math.h>


/* function prototypes */
double velocity(double), func(double);
double bisect(double (*)(double), double,
              double, double);

#define TOLERANCE .00001 /* tolerance for convergence */
#define CLOSE  .001      /* check if root found */
```

The symbols **TOLERANCE** and **CLOSE** are defined for use later in the program. **TOLERANCE** is the tolerance value used in the bisection method to determine whether the change in the estimate of the root is small between iterations. The value **CLOSE** will

be used to test whether the value returned by **bisect()** is "close enough" to zero to be acceptable as a root.

The function **velocity()** is a C version of the function describing the velocity of the train over time. The function **func()** is a C implementation of $85 - v(t)$. The computation of the train's velocity could be integrated into the implementation of **func()** to reduce both the amount of code and the computational time. We leave them separate here for clarity.

The function **bisect()** is the root-finding function given in Section 3.5.2. This code is not reproduced here. We assume that the code for the function **bisect()** is in the same source code file as our code below.

The remainder of the program is the code for the functions **main()**, **velocity()**, and **func()**.

```
/* main function for train acceleration problem */

main()
{
  double root;

  root = bisect(func,  0.0, 100.0, TOLERANCE);

  if(fabs(func(root)) < CLOSE)
    printf("Max speed reached in %7.0lf secs.\n",
            floor(root+.5));
  else

    printf("Value %lf not close to root.\n", root);
}

/* compute velocity as function of time */

double velocity(double t)
{

  return(29.4+.795*t+1.0/(1.0-1.034*exp(.0061667*t)));
}

/* function computes 85 minus  velocity of train  */

double func(double t)
{
  return(85.0-velocity(t));
}
```

Note that after we invoke **bisect()** from **main()** we check whether the value **root** returned is in fact within **CLOSE** of a true root. This is to avoid the possibility of false convergence in the root-finding method.

We also use the math library function `floor()` to round off the value of `root` to the nearest integer. The expression `floor(root+.5)` adds one half to `root` and then finds the largest `double` which is less than `root+.5` and is representable as an integer. This has the effect of rounding `root` to the nearest integer.

The result output by this program is that, to the nearest second, the maximum speed is reached in 72 seconds. The bisection method finds a true root (or more accurately, a value close enough to the true root) for this case.

3.7 NUMERICAL INTEGRATION

A second type of commonly encountered mathematical problem is solving for the value of some definite integral. In most calculus courses, students are taught a range of analytic methods for solving integrals. However, there are many integrals which are inherently unsolvable. In the introduction to this chapter, we discussed the fact that the integral of the normal (or Gaussian) distribution has no closed-form, analytic solution. Stated somewhat more formally, the value of the integral

$$\int_a^b \frac{1}{\sqrt{2\pi}} e^{-x^2/2}\, dx$$

has no closed-form solution. Numerical methods are used to evaluate this integral for specific values of a and b.

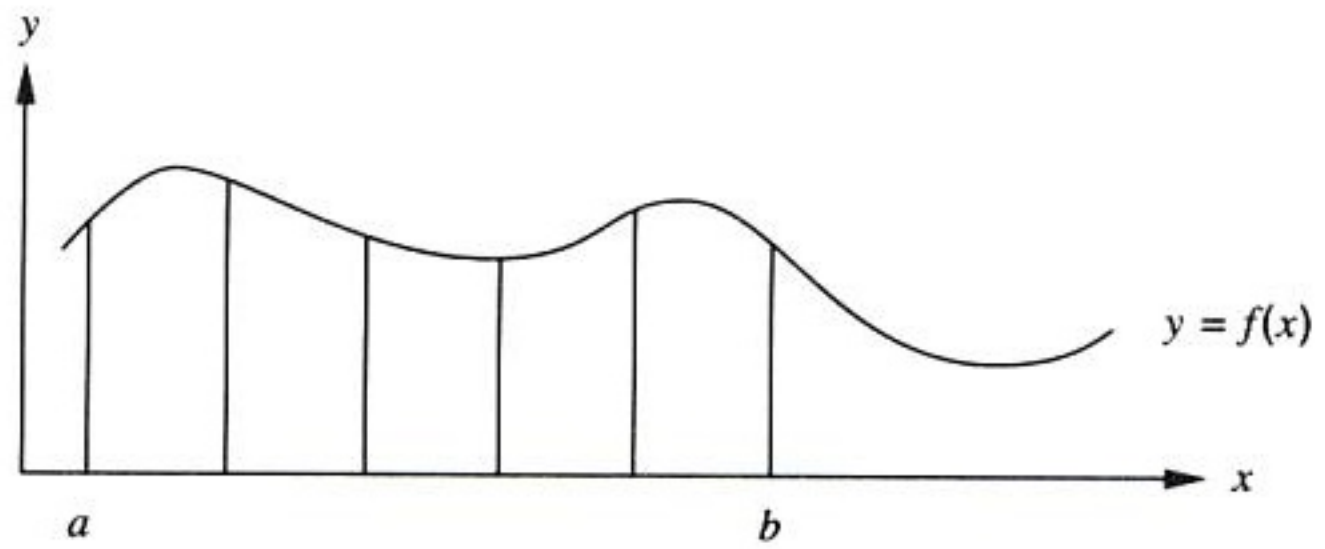
3.7.1 Rectangular Rule

The simplest numerical integration technique is called the *rectangular* method. The x axis is divided into n equal segments in the interval $[a, b]$, creating a series of panels as shown in Figure 3.9a. The area under the function $f(x)$ is approximated by the sum of the areas of the rectangles, the heights of which are either the left sides or the right sides of the panels. Figure 3.9b and c depict the areas for the left- and right-side rules, respectively. Clearly, the use of the two different sides will produce different approximations for the integrals.

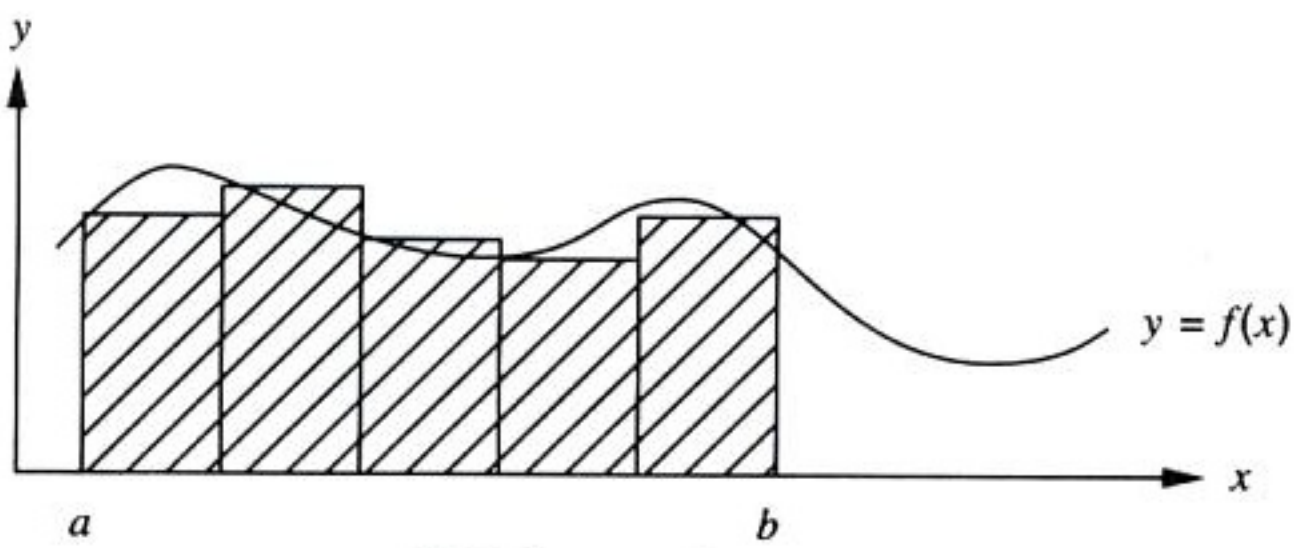
The rectangular method is rarely used in practice because alternative approaches that are considerably more accurate involve only a little more complexity. For this reason, we will not show the implementation of the rectangular method in C.

3.7.2 Trapezoidal Rule

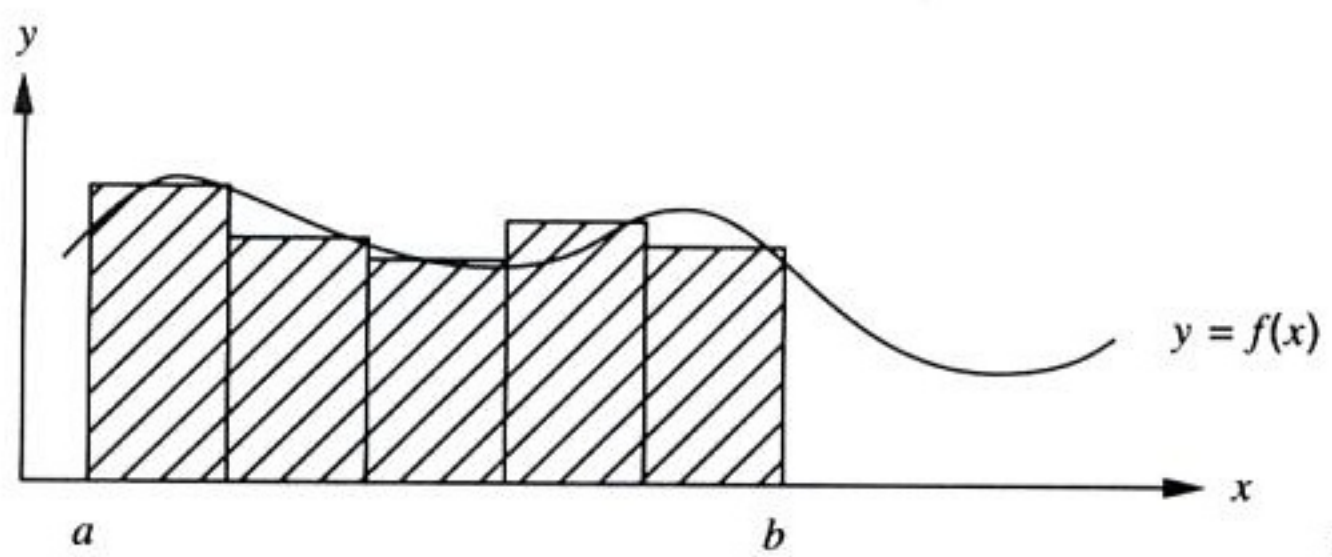
One way to improve on the rectangular method for solving integrals numerically is to divide the x axis into n equal intervals and to approximate the area under the function as the sum of the areas of trapezoids. Not surprisingly, this technique is called the



(a) Function to be integrated



(b) Left rectangle method



(c) Right rectangle method

Figure 3.9 Graphical interpretation of the rectangle method.

trapezoidal method. The trapezoidal method requires almost the same computation as the rectangular method and for a fixed number of subdivisions of the x axis yields more accurate approximations. Figure 3.10 depicts the approximate areas computed with this method.

The reader can verify that the trapezoidal method is equivalent to averaging the results from two versions of the rectangular method, one using the right side of each panel to determine the height of the rectangle and the other using the left side. However, computing the trapezoidal approximation this way is extremely inefficient because it

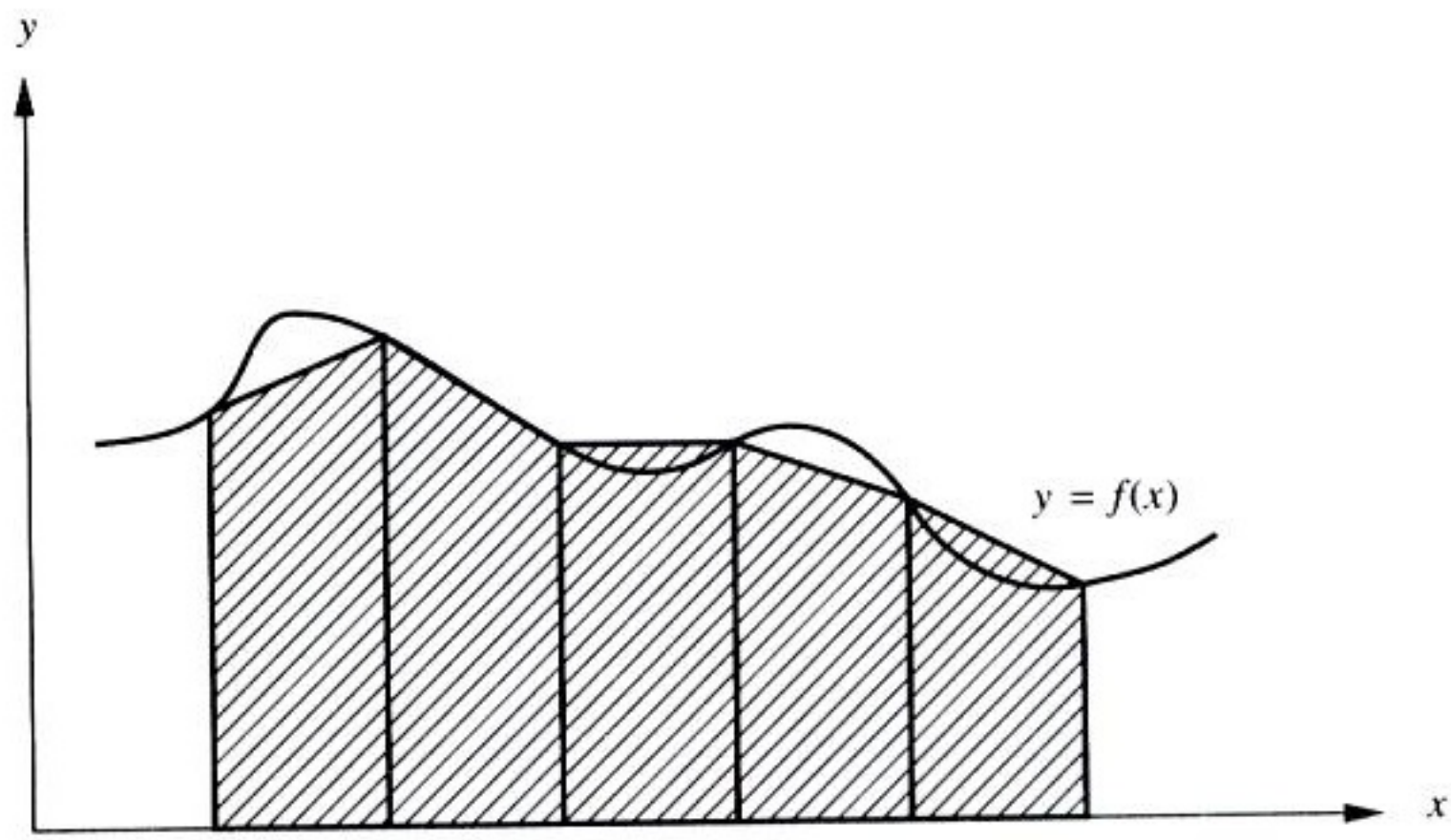


Figure 3.10 Graphical interpretation of the trapezoidal method.

requires many more calls to evaluate the function $f(x)$ than is needed. A more efficient implementation of the trapezoidal rule in C is as follows:

```
/* implementaton of trapezoidal method */
double trapezoidal(
  double (*f)(double),   /* function to be integrated */
  double a,              /* starting x */
  double b,              /* ending x   */
  int n)                 /* number of panels */
{
  double answer, h;      /* result and panel width */
  int i;                 /* counter for intervals */

  answer = f(a)/2;
  h = (b-a)/n;

  /* sum panel areas */
  for(i=1; i<=n; i++)
    answer = answer + f(a+i*h);

  answer = answer - f(b)/2;
  return(h*answer);
}
```

The function prototype for `trapezoidal()` is

```
double trapezoidal(double (*)(double),
                   double, double, int);
```

In implementing the trapezoidal rule, the two endpoints a and b must be handled separately because they are used for computing the area under only one interval, while the other points $f(x+h)$, $f(x+2h)$, ..., $f(x+(n-1)h)$ are each used in approximating the area under two intervals. The function `trapezoidal()` deals with the endpoints by initializing `answer` to `f(a)/2` and subtracting `f(b)/2` from `answer` before it is multiplied by `h`.

3.7.3 Simpson's Rule

A considerable improvement over the trapezoidal rule can be achieved by approximating the function within each of the n intervals by some polynomial. When second-order polynomials are used, the resulting algorithm is called *Simpson's rule*.

If we define h as the width of each of the subdivisions on the x axis and apply Simpson's rule, then the area under the curve in the interval $(x, x+h)$ is approximated by

$$\frac{h}{6}\left[f(x) + 4f\left(x + \frac{h}{2}\right) + f(x+h)\right]$$

This mathematical result is used in the implementation of Simpson's rule given below.

```
/* implementation of Simpson's rule */
double simpson(
  double (*f)(double),  /* function to be integrated */
  double a,             /* starting x */
  double b,             /* ending x   */
  int n)                /* number of panels */
{
  double answer, h;     /* result and panel width */
  double x;
  int i;

  answer = f(a);
  h = (b-a)/n;

/* sum panel areas */
  for(i=1; i<=n; i++) {
    x = a+i*h;
    answer = answer + 4*f(x-h/2) + 2*f(x);
  }
  answer = answer - f(b);
  return(h*answer/6);
}
```

It is possible to derive approximations that involve higher-order polynomials. However, unless the function $f(x)$ is particularly burdensome to compute, it is often simpler to increase the number of intervals rather than implement more complicated approximations.

3.7.4 Adaptive Integration

The methods described above all require the user to select a value for the number of intervals, n. This shortcoming can be alleviated by starting with a small number of intervals and successively doubling that number until the change in the approximated integral is small by some measure. This method relies on the hypothesis that the error in the integral approximation decreases as n increases. While this is not always true, it is a reasonable working hypothesis for most functions.

Below is a simple (but not particularly efficient) implementation of this procedure for Simpson's rule.

```
#include <math.h>
double new_simpson(
  double (*f)(double),   /* function to be integrated */
  double a,              /* starting x */
  double b,              /* ending x   */
  int n0,                /* number of panels */
  double tolerance)      /* measure of convergence */

{
  double check= tolerance +1.0;
  double lowval, val;

  lowval = simpson(f, a, b, n0);
  while(check  > tolerance) {
    n0 = 2 * n0;
    val = simpson(f, a, b, n0);
    check = fabs((val-lowval)/val);
    lowval = val;
  }
  return(val);
}
```

A more serious shortcoming of the integration methods we have described is that they use the same interval width over the entire integral. A better procedure would be to use a wide interval on portions of the function that are nearly linear (when the trapezoidal rule is being used) or nearly quadratic (when Simpson's rule is being used), and smaller intervals for portions of the function that are more complicated. A technique

which can do that is called *adaptive* because its adapts the interval width to the shape of the function. Adaptive methods offer the potential for reducing computation costs and increasing the accuracy of numerical methods. In this subsection we explore only the simplest forms of adaptation.

Our adaptive integration method begins with a small number of intervals. Define n_0 to be that initial number. It then uses a nonadaptive integration method to integrate the function with both n_0 intervals and $2n_0$ intervals. If the difference between the results of these two approximations is small, the algorithm returns the result using $2n_0$ intervals. If, however, the difference exceeds some tolerance, then the initial interval (a, b) is divided into two intervals of equal size. The adaptive method is then applied on both halves, and the results are added together.

This procedure is intrinsically recursive. Each half interval may be divided further depending on whether such subdivision is needed. Below is an implementation of this method that uses the function **simpson()**.

```
#include <math.h>
double adapt(
  double (*f)(double),   /* function to be integrated */
  double a,              /* starting x */
  double b,              /* ending x   */
  int n,                 /* number of panels */
  double tolerance)      /* measure of convergence */

{
  double val, check;
  val = simpson(f,a,b,2*n);

  check = fabs((simpson(f,a,b,n)-val)/val);
  if(check  > tolerance)
    val = adapt(f,a,(a+b)/2.0,n,tolerance) +
            adapt(f,(a+b)/2.0,b, n,tolerance);
  return(val);
}
```

3.7.5 Other Methods

All of the algorithms for numerical integration presented in this section are what are usually called *classical methods*. They rely on dividing the x axis in some regular way. There are other numerical integration algorithms called *quadrature methods*, that are beyond the scope of this book. The reader is referred to Press et al. [Press, William H. 88] for a detailed discussion of these widely used techniques. Another method, called *Monte Carlo integration*, is described in Chapter 8.

As expected, the trapezoidal method works well when **a** and **b** are small but produces large errors for unit intervals with higher values of **a** and **b**. The adaptive method is accurate to at least five significant digits for all the intervals tested. Simpson's method works better than the trapezoidal method but also begins to fail for the later intervals tested.

3.9 ORDINARY DIFFERENTIAL EQUATIONS

Note to Reader: The reader of this section is assumed to be familiar with differential calculus. The section can be skipped without loss of continuity.

A vast range of problems in science and engineering are characterized by *differential equations*, where the rate of change of some quantity is a function of the values of other variables, potentially including other rates of change. It is quite common for these differential equations to be mathematically unsolvable, requiring that numerical methods be applied.

We restrict our presentation here to the solution of first-order ordinary differential equations. These are of the form

$$\frac{dy}{dx} = g(x, y)$$

or, defining $y' = dy/dx$,

$$y' = g(x, y)$$

Restricting attention to only first-order differential equations is not as limiting as it first may seem. It is quite simple to convert an mth-order differential equation into a system of m first-order equations. Thus, with minor extensions all the methods we explore below allow for solution of ordinary differential equations of arbitrary order.

The simplest way to transform an mth-order equation to a system of first-order equations is to define a new variable for each of the derivatives of y up to order $(m - 1)$. Suppose that the original differential equation is of the form

$$y''' = g(x, y, y', y'')$$

Then we can define

$$y_0 = y$$

$$y_1 = y'$$

$$y_2 = y''$$

The original third-order differential equation can now be written as

$$y_2' = g(x, y_0, y_1, y_2)$$

$$y_1' = y_2$$

$$y_0' = y_1$$

In order to keep the notation and examples simple, we focus mostly on single equations in this chapter. However, as illustrated in Section 3.10, the methods we present easily generalize to systems of first-order differential equations.

The solution of any differential equation is not unique unless there is a set of constraints. We consider here only initial value problems, where the constraints are the values of $y_0, y_1, \ldots, y_n$ at some initial value of the independent variable x. We will define the initial values of x and y to be x_0 and y_0, respectively. Solution of systems of ordinary differential equations with other boundary conditions is considerably more complicated. The reader is referred to Press et al. [Press, William H. 88].

The numerical algorithms described in this section do not really "solve" differential equations in the mathematical sense. Rather, they compute a series of points that are numerically close to the solution. The function $y = f(x)$ that satisfies the differential equation remains unknown. For most applications, knowing a sufficient number of points on the function $y = f(x)$ is adequate.

3.9.1 Euler's Method

The simplest numerical algorithm for solving differential equations is called *Euler's method*. This method is by far the most straightforward, but unfortunately is inaccurate. For this reason it is rarely used in practice. We present it here because it serves as the basis for more accurate algorithms.

Euler's algorithm can be summarized as follows:

- **Step 1.** Define the following variables:
  `x` = x_0, the initial value of x
  `y` = y_0, the initial value of y
  `xlast`, the largest value of x for which a solution is desired
  `h`, an increment of x
- **Step 2.** Output `x` and `y`.
- **Step 3.** Set `x = x + h`.
- **Step 4.** Compute `y = y + h*g(x,y)`.
- **Step 5.** If `x > xlast`, terminate. Otherwise, go to step 2.

The following C function implements Euler's method. Its arguments are:

`x0` and `y0` — initial values for x and y.
`g` — a pointer to the function $g(x, y)$ used to compute the derivative.
`h` — a step size.

xlast — the largest value of x for which points along the equation are to be computed.

```
/* this function implements Euler's method */

#include <stdio.h>

void euler(
  double x0,      /* Initial x */
  double y0,      /* Initial y */
  double (*g)(double,double),
            /*pointer to derivative function */
  double h,       /* increment */
  double xlast)   /* largest x */

{

  printf("        x                    f(x)\n");
  for ( ; x0 <= xlast; x0 = x0+h) {
    printf("%15.6g  %15.6g\n", x0, y0 );
    y0 = y0 + h * g(x0, y0);
  }
}
```

The implementation of Euler's method above takes advantage of the fact that C passes arguments by value. This makes it possible for the function to use the arguments **x0** and **y0** without affecting the values of the variables used in the function's invocation.

Using the function **euler()** in a program requires coding the function $g(x, y)$. For example, if the differential equation to be solved is

$$y' = -\frac{y-4}{x}, \qquad y(1) = 0$$

then the C function for the differential equation would be

```
double func(double x, double y)
{
  return( -(y-4)/ x );
}
```

With the code above, the function **euler()** could be used to find the path of the solution for x going from 1 to 4 in steps of .1 as follows:

```
/* function prototypes */

void euler(double, double, double (*)(double, double),
                   double, double);
```

```
double func(double, double);

/* invocation of Euler's method */

euler(1.0, 0.0, func, .1, 4.0);
```

3.9.2 Sources of Error in Euler's Method

Euler's algorithm uses $g(x, y)$ repeatedly to compute the derivative of y with respect to x, and then applies that derivative to take a step along the x axis. This is shown graphically in Figure 3.12. In that figure, the solid line represents the true solution to the differential equation, while the line connecting the boxes shows the path computed by Euler's algorithm.

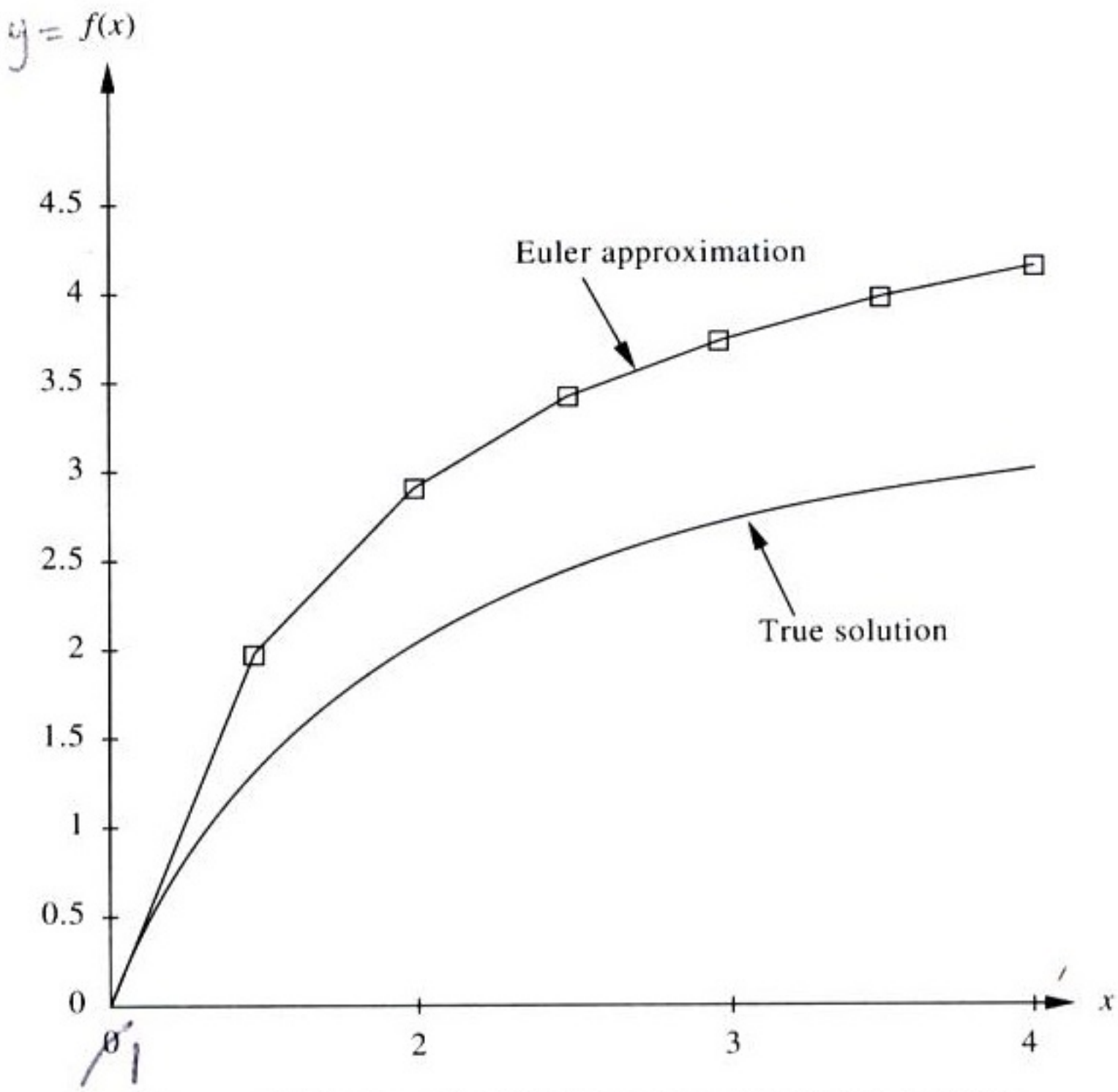


Figure 3.12 Graphical interpretation of Euler's method.

The first step in Euler's method moves along the tangent to the true solution of the differential equation by `h` along the x axis and `h*g(x,y)` on the y axis. This produces a point $(x_0 + h, y_0 + hg(x_0, y_0))$. This point differs from the true solution by some amount Δy_1, the "local" error. This error can be shown to be of order h^2, where h is assumed to be small. We use the notation $O(h^2)$ to denote that the error is of order h^2.

In the second and subsequent steps, the local error is only one source of error. The second source is the fact that the derivative is being computed at a point which is different from the true solution to the differential equation. The size of this second error depends crucially on how close the computed derivative is to the true derivative, which in turn depends on the peculiarities of the specific differential equation being solved. Some differential equations are unstable in the sense that small deviations from the true solution result in very large differences in the derivative. The use of Euler's method on these equations can produce huge errors after a relatively small number of steps.

Consider, for example, the unstable differential equation

$$y' = 6y - 8e^{-2x}, \qquad y(0) = 1$$

The analytic solution to this differential equation is

$$y = e^{-2x}$$

HW!

However, the function

$$y' = e^{-2x} + ce^{6x}$$

is a general solution applicable to other initial conditions. As local errors in Euler's method accumulate, the second term of the general solution contributes significantly to subsequent steps of Euler's method, changing the derivative in each step away from the true value. These errors accumulate over the steps, leading to entirely erroneous results. Even with very small step sizes, Euler's method will diverge from the true solution quite quickly.

3.9.3 Runge-Kutta Methods

Because of the relatively large local error and the shortcomings of Euler's method for differential equations that are unstable (or nearly so), the method is rarely used in practice. However, the basic idea of Euler's method can be adapted to produce a far more robust and useful algorithm. These adaptations are grouped together in a class of algorithms called *Runge-Kutta methods*.

Runge-Kutta methods are based on the observation that each step of Euler's method relies entirely on the derivative evaluated at a single point. If one could compute a better estimate of the derivative, it would be possible to improve on each Euler method step. Runge-Kutta methods do this by evaluating the derivative at more than one point for each step and combining the various computed derivatives to obtain a better approximation of the solution. The goal is to reduce the local error at each step to something considerably smaller than $O(h^2)$.

The simplest variant of Runge-Kutta methods uses the same derivative as used in Euler's method and a derivative computed at the point $x + h/2$, the midpoint of the interval spanned by a single Euler method step. The algorithm for this is as follows:

- **Step 1.** Define the following variables:
  `x` = x_0, the initial value of x
  `y` = y_0, the initial value of y
  `xlast`, the largest value of x for which a solution is desired
  `h`, an increment of x
  `temp`, a temporary variable
- **Step 2.** Output `x` and `y`.
- **Step 3.** Compute `temp = h*g(x,y)`.
- **Step 4.** Compute `y = y + h*g(x + h/2, y+temp/2)`.
- **Step 5.** Set `x = x + h`.
- **Step 6.** If `x > xlast`, terminate. Otherwise, go to step 2.

In this algorithm, the derivative at the midpoint between x and $x + h$ is used to compute the direction for the next step. This reduces the local error per step from $O(h^2)$ in Euler's method to $O(h^3)$. This is called the *second-order* Runge-Kutta method. (Euler's method can be viewed as the first-order Runge-Kutta method.)

Higher-order Runge-Kutta methods involve more complicated combinations of intermediate derivatives. By appropriately combining additional estimates of the derivative, we can reduce the order of the error term to still higher powers of h. Solutions for the appropriate weights for the various terms are beyond the scope of this book.[3] Instead, we state the results for one of the most widely used Runge-Kutta variants, the fourth-order method. This is given in the following algorithm.

- **Step 1.** Define the following variables:
  `x` = x_0, the initial value of x
  `y` = y_0, the initial value of y
  `xlast`, the largest value of x for which a solution is desired
  `h`, an increment of x
  `ta, tb, tc,` and `td`, temporary variables
- **Step 2.** Output `x` and `y`.
- **Step 3.** Compute:
  ```
  ta =  h*g(x, y)
  tb =  h*g(x+h/2, y+ta/2)
  tc =  h*g(x+h/2,y+tb/2)
  td =  h*g(x+h, y+tc)
  ```
- **Step 4.** Compute `y = y + ta/6 + tb/3 + tc/3 + td/6`.
- **Step 5.** Set `x = x + h`.
- **Step 6.** If `x > xlast`, terminate. Otherwise, go to step 2.

[3]Readers are referred to Abramowitz and Stegun [Abramowitz, Milton 64].

An implementation of this algorithm in C is as follows:

```
/* fourth-order Runge-Kutta method */
void runge_kutta(
  double x0,      /* Initial x */
  double y0,      /* Initial y */
  /*pointer to derivative function */
  double (*g)(double,double),
  double h,       /* increment */
  double xlast)   /* largest x */
{

  double ta, tb, tc, td;
  printf("        x                f(x)\n");
  for ( ; x0 <= xlast; x0 = x0+h) {
    printf("%15.6g %15.6g\n", x0, y0 );
    ta =  h * g(x0, y0);
    tb =  h * g(x0+h/2.0, y0+ta/2.0);
    tc =  h * g(x0+h/2.0,y0+tb/2.0);
    td =  h * g(x0+h, y0+tc);
    y0 = y0 + (ta +2.0* tb +2.0*tc + td)/6.0;

  }
}
```

The differences between the accuracy of a fourth-order Runge-Kutta method and Euler's algorithm can be seen by comparing them on a simple problem with a large step size. Consider the following differential equation:

$$y' = -\frac{y-4}{x}, \qquad y(1) = 0$$

The true solution to this equation is

$$y = 4 - \frac{4}{x}$$

If we use Euler's method to solve this equation in the interval $1 \le x \le 4$ using a step size of .1, we find the errors depicted in Figure 3.13. This figure plots the difference between the function computed by Euler's method and the true function. If we apply the Runge-Kutta method to this same problem with an identical step size, the absolute value of the errors are always less than 1.2×10^{-16}. Thus, even with a relatively large step size, the Runge-Kutta method produces very accurate results on this problem, while the errors in Euler's method are significant.

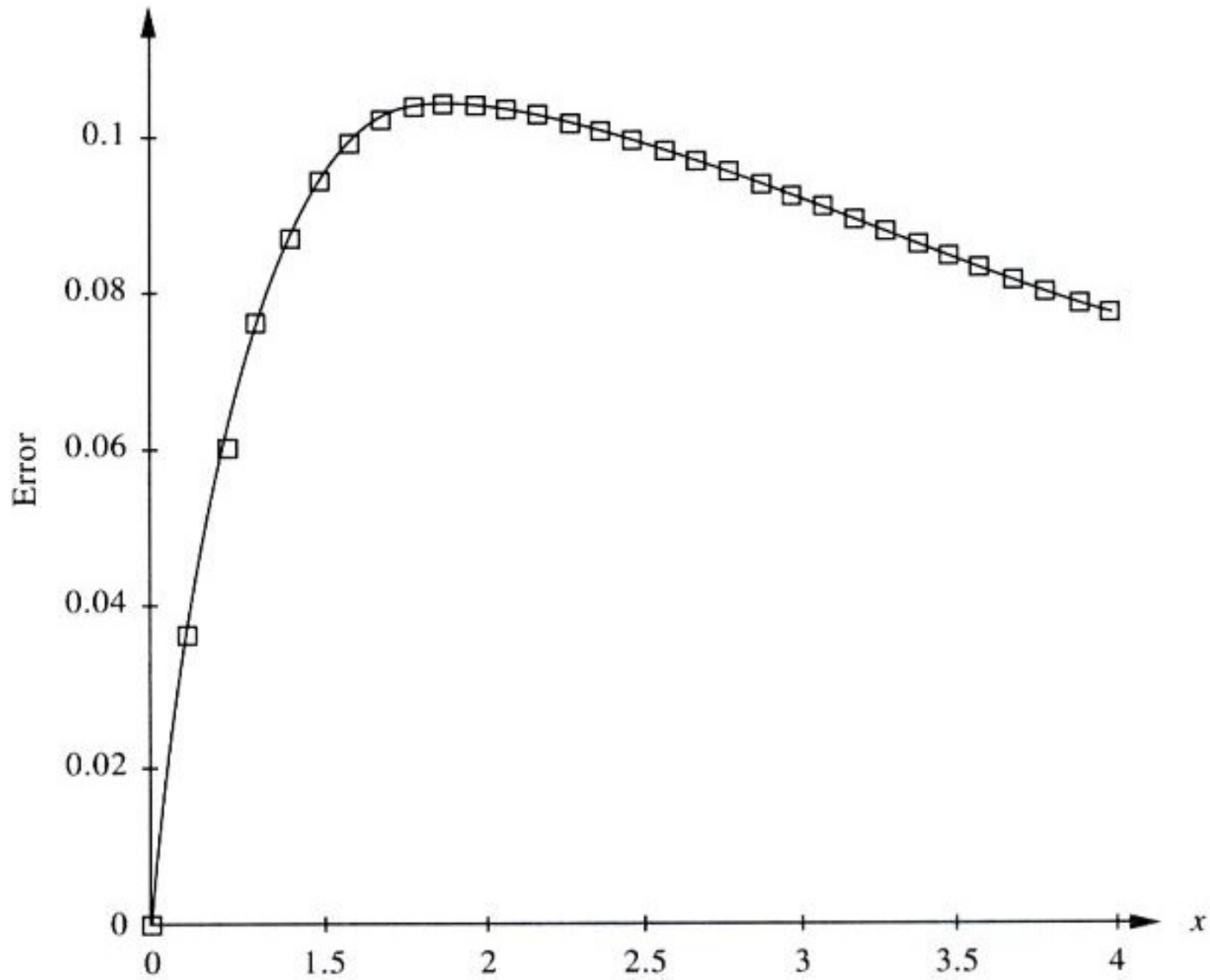


Figure 3.13 Example of errors in Euler's method.

3.10 PROGRAMMING PROJECT: MOTION OF A PENDULUM

We now consider the application of the Runge-Kutta method to solving for the motion of a pendulum. The pendulum consists of a mass at the end of a rigid rod of length l that is attached to a frictionless pivot as shown in Figure 3.14. We assume for simplicity that the weight of the rod is negligible compared to the mass.

Figure 3.14 shows the geometry of the pendulum's motion. We call s the *arc length* as measured along the circle of radius l centered at the pivot. The value $s = 0$ corresponds to the position of the pendulum at rest. If moved away from the rest position, the mass will move along the circle, changing the value of s. We denote that path of the mass over time by the function $s(t)$.

From the geometry shown in Figure 3.14, we can show that $s = l\theta$. If we use a rectangular coordinate system centered at the pivot of the pendulum, we can also show that

$$x(t) = l \sin\theta(t) = l \sin\frac{s(t)}{l}$$

$$y(t) = -l \cos\theta(t) = -l \cos\frac{s(t)}{l}$$

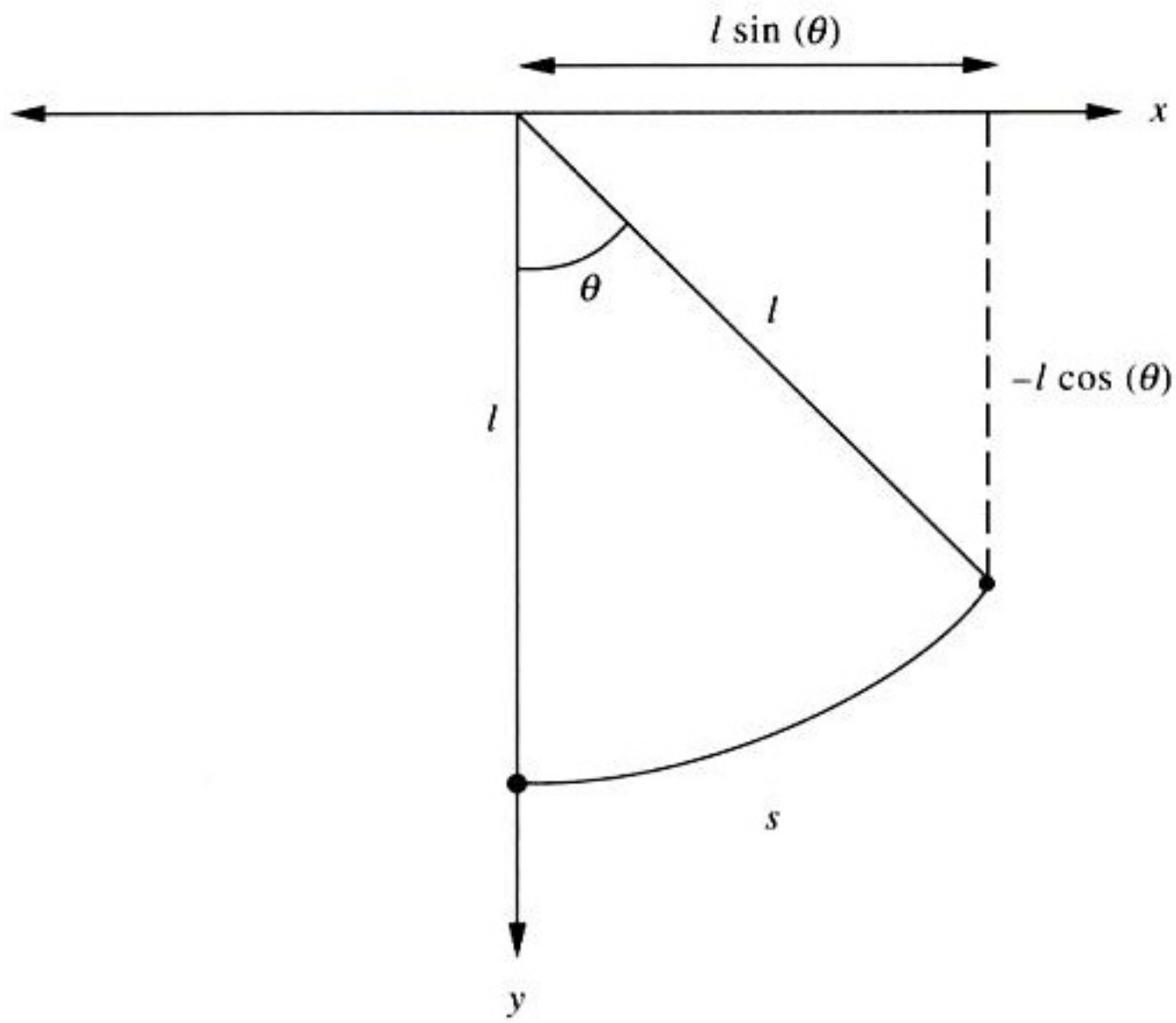


Figure 3.14 Geometry of a simple pendulum.

We assume that the pendulum starts from an initial position $s(0) = s_0$. By applying Newton's law of gravitation and the equation that force equals mass times acceleration, it can be shown that the movement of the pendulum is governed by the differential equation

$$\frac{d^2s}{dt^2} = -g \sin\frac{s(t)}{l}$$

where g denotes the gravitational constant (9.8 meter/sec^2). Once we solve for $s(t)$, the movement of the pendulum in the (x, y) coordinate system can be computed from the geometric relationships above.

It is common in introductory physics texts to make the assumption that movements of the pendulum are small, and that therefore the value of $\sin\theta$ can be approximated by θ. This makes analytic solution of the differential equation easier. Our approach in this section will be to keep the correct differential equation and solve for the path of the pendulum over time numerically using the Runge-Kutta algorithm.

Since the differential equation has a second derivative term, we will solve it as a system of two first-order differential equations by introducing the function $z(t)$ defined as the derivative of $s(t)$ with respect to t. Using this intermediate function, the movement of the pendulum can be described by a system of two first-order differential equations given as follows:

$$\frac{dz(t)}{dt} = -g \sin\frac{s(t)}{l}$$

$$\frac{ds(t)}{dt} = z(t)$$

The following C program adapts the fourth-order Runge-Kutta function developed in Section 3.9.3 to solve this system of two first-order differential equations. Each step of the algorithm will require keeping track of two dependent variables, s and z. As with the single-variable case, we will update the variables in four substeps. The program will output the values of $s(t)$, $x(t)$, and $y(t)$. The C code is as follows.

```
/* program to solve for movement of a pendulum */

#include <stdio.h>
#include <math.h>
#define GRAVITY -9.8  /* gravitational constant */

main()
{
  double h;        /* time increments for computation */
  double s0;       /* initial  pendulum position */
  double tlast;    /* time limit for computation */
  double length;   /* length of pendulum */
  double s,z,t;            /* values of s,z and t */
  double sa, sb, sc, sd; /* intermediate values of s */
  double za, zb, zc, zd; /* intermediate values of z */

/* prompt for and read input values */
  printf("Enter initial value of s:");
  scanf("%lf", &s0);
  printf("Enter last value of t:");
  scanf("%lf", &tlast);
  printf("Enter value of time increments:");
  scanf("%lf", &h);
  printf("Enter length of pendulum:");
  scanf("%lf", &length);

/* print labels */
  printf("    t      s(t)      x(t)      y(t)\n");

/* initialize values */
  s = s0;
  z = 0;

/* perform runge-kutta iterations */
  for (t=0 ; t <= tlast; t = t+h) {

  /* output the values of key variables */
    printf("%12.6g  %12.6g %12.6g %12.6g\n", t, s,
      length*sin(s/length), -length*cos(s/length));
  /* update z and s */
```

```
        za = h * GRAVITY * sin(s/length);
        sa = h * z;
        zb = h * GRAVITY * sin((s+sa/2)/length);
        sb = h * (z+za/2);
        zc = h *GRAVITY * sin((s+sb/2)/length);
        sc = h * (z + zb/2);
        zd = h * GRAVITY * sin((s+sc)/length);
        sd = h * (z + zc);
        z = z + za/6 + zb/3 + zc/3 + zd/6;
        s = s + sa/6 + sb/3 + sc/3 + sd/6;
    }
}
```

3.11 SUMMARY OF CHAPTER 3

In this chapter we introduced the key ideas in using computers to solve numerical problems. The methods are applicable to a broad range of problems encountered in science, engineering, management, and the quantitative social sciences.

The representation of floating-point values in a modern, digital computer was described. While different computers may use different representations, all computers separate a floating-point number into a sign indicator, a fraction part, and an exponent. The range of values which can be represented and the precision of any floating-point number varies across machine types.

Despite this variation, all computers that use floating-point numbers share the characteristic that not all numerical values can be represented exactly. As a consequence, all arithmetic operations involving floating-point numbers must be viewed as introducing at least some small error. Measured as a fraction of the result of a floating-point computation, this error is at least as large as the *machine precision*, the largest number which when added to 1.0 does not change the value 1.0.

Floating-point errors can accumulate as the results from one floating-point operation are used in subsequent operations, resulting in numerical answers that can be considerably in error. Programmers must be aware of the potential for floating-point errors, and must design their algorithms to minimize the effects of such errors.

In the chapter we described some of the methods used for finding a root of an equation. Writing this type of algorithm in general form required the ability to pass functions as arguments to other functions. This is accomplished in C by passing a *pointer to a function* as an argument.

The particular root-finding methods described are *bisection*, the *secant method*, and *Newton's method*. Bisection and the secant method require an interval known to contain a root as an input, while Newton's method requires only an initial guess of the root. Neither method offers any guarantee that all roots of the function will be found.

Methods for numerical integration were then introduced. The approaches developed included the rectangular, trapezoidal, and Simpson's rules. Techniques that allow the

CHAPTER

4

LINEAR ALGEBRA

4.1 INTRODUCTION

In this chapter we explore numerical methods that make use of C's ability to represent arbitrary collections of elements of the same type as a single unit. These collections are called *arrays*.

One use of arrays we will explore extensively is their use in solving problems in *linear algebra*. These methods are important because of the wide range of real-world problems which can be approximated by linear systems of equations. In addition, nonlinear problems are often solved by methods which rely on the ability to solve linear approximations. For example, Newton's method for finding a root of an equation, described in Chapter 3, relies on successive linear approximations.

In order to develop functions that solve key problems in linear algebra, we will need a way to represent *vectors* and *matrices*, the key building blocks of linear systems. C (and virtually all other general-purpose programming languages) provides a straightforward way to represent such objects.

Arrays have uses that go well beyond their application in linear algebra. In this chapter we demonstrate their usefulness in diverse areas, including population forecasting, the analysis of transportation networks, and the simulation of one computer in a program executing on another computer.

We begin by introducing single-dimension arrays in C as a way to represent vectors. We demonstrate how functions that manipulate single-dimension arrays can be written, using simple operations such as the dot product of two vectors as examples. We also develop a complete population forecasting model as a programming project.

We then consider the more general case of representing matrices using two-dimensional arrays. This provides the needed background for developing more complicated algorithms, such as solving systems of linear equations and inverting a matrix.

The final two sections of the chapter are programming projects that show two very different uses of arrays. The first project solves what is called the *all-pairs shortest path problem*, in which a transportation or other network linking various points is analyzed to find the shortest way to get from every point to every other point. The second project develops a simulator for the simple robot introduced in Chapter 1.

4.2 REPRESENTING VECTORS IN C

4.2.1 What Are Arrays?

C

Suppose that we wanted to represent a group of data points corresponding to some measurements. For example, we might have a series of measurements on the level of carbon monoxide in the air for n different days, and we might want to take the average of the data. If there are n such values $x_0, x_1, \ldots, x_{n-1}$, we could define a separate variable for each measurement. If there were seven such measurements, we might have the definition

```
float x0, x1, x2, x3, x4, x5, x6;
```

The average could then be computed as

```
float average;
average = (x0+x1+x2+x3+x4+x5+x6)/7.0;
```

While this approach might work on data for just seven days, it would be unmanageable if we had data for thousands of measurements. In addition to having to define a huge number of separate variables, we would have to code a gigantic summation. Even worse, every time we added yet another observation, we would have to modify the code and recompile it. Clearly, a very different approach is needed.

The C language provides such a different approach by allowing the programmer to collect groups of variables of the same type into a single "object" called an *array*. For example, if we had 1000 observations, we could define a single array called **x** that had 1000 **float**s. This would be defined in the statement

```
float x[1000]; /* x is an array with 1000 entries */
```

The square brackets ([and]) indicate that **x** is an array, and the integer constant **1000** denotes that the array has 1000 entries. C allows definition of arrays of any standard type. There are no fixed limits on how many entries a single array may have, though some C implementations do enforce specific restrictions.

Once we have defined an array, we can reference any element of the array as **x[ex]**, where **ex** is an expression that evaluates to some integer data type. The expression is called the *subscript*. In C, the first entry in any array **x** is **x[0]**, and the elements in the array with **n** elements are referenced as **x[0]**, **x[1]**, **x[2]**, . . . , **x[n-1]**. For example, we could read data into an array with the following statements:

```
/* read 1000 values into array x  */
float x[1000];
int i;
for(i=0; i<1000, i++)
  scanf("%f", &x[i]);
```

Note that the second argument to the function **scanf** in this example is **&x[i]**. This relies on the fact that the precedence of the subscript operator (denoted by the square brackets) is higher than that of the "address of" operator.

Similarly, the average of the 1000 values could be computed as

```
/* compute average of 1000 values in array */
float x[1000];
float average = 0.0;

for(i=0; i<1000; i++)
  average = average + x[i];
average = average/1000;
```

One way to view arrays is to visualize them as a block of adjacent memory locations. The beginning of the array (the zeroth element) is the first location in the block, the next location holds element one of the array, etc. Figure 4.1 depicts a section of a computer's memory and the elements of an array that has **n** values. Note that the figure shows array elements in sequential memory locations. Depending on the type of array (**char**, **int**, etc.), each array element may occupy more than one memory location. For example, on many computers, a **double** occupies eight bytes, so an array of 10 **double**'s would occupy 80 bytes in total.

4.2.2 Arrays, Pointers, and Address Arithmetic

Arrays and pointers are closely related in many ways. Perhaps the most important relationship is that *the name of an array is a synonym for the address of its zeroth element*. For example, the C fragment

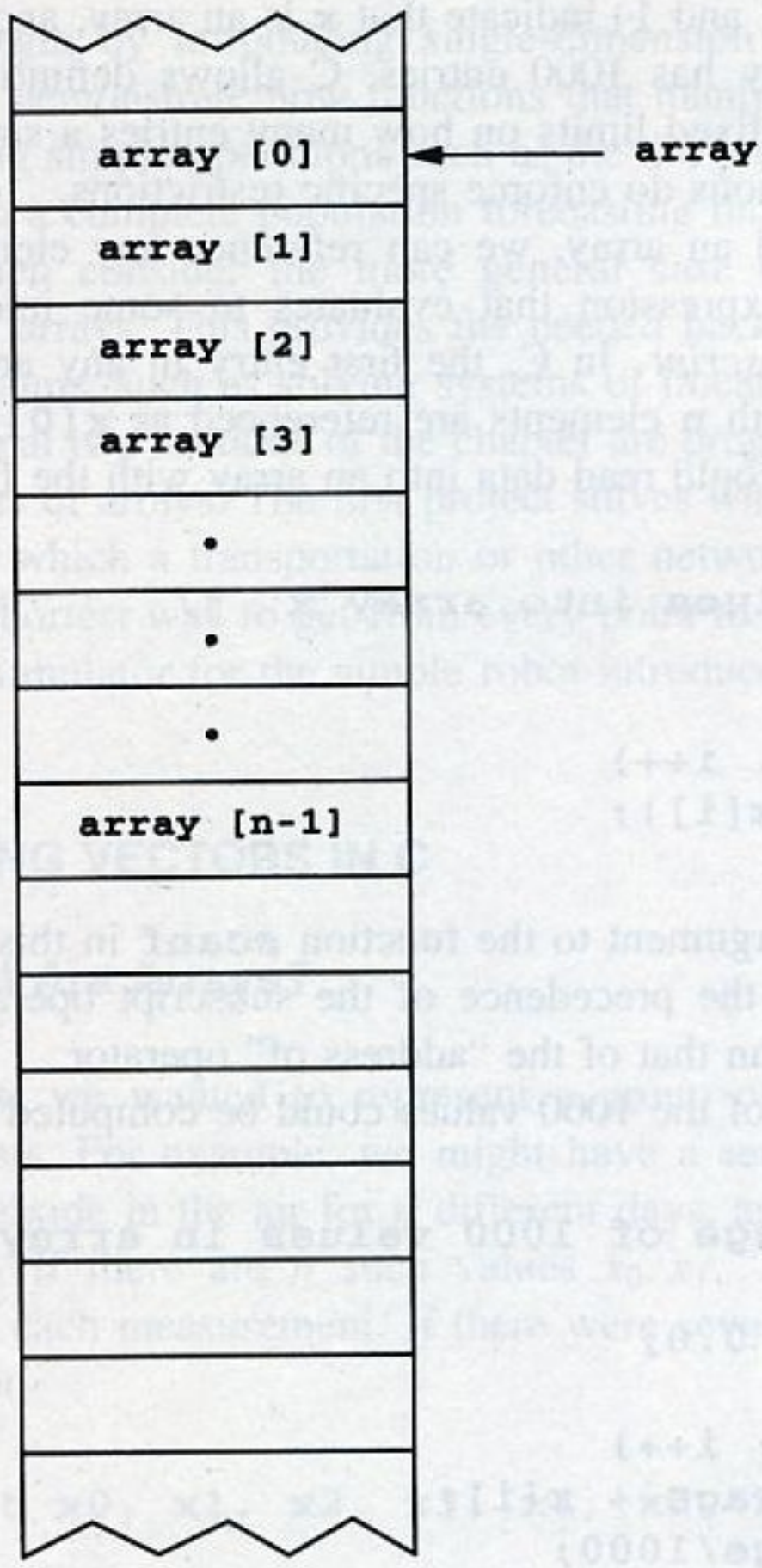


Figure 4.1 Arrays in memory.

```
int  values[1000];
int *pvalues;
pvalues = &values[0];
```

declares **values** to be an array of **int**s and **pvalues** to be a pointer to an **int**. The fragment then assigns **pvalues** the address of the zeroth element of **values**.

What is less obvious is that the assignment

```
pvalues = &values[0];
```

is entirely equivalent to the statement

```
pvalues = values;
```

This is because the name of an array is a synonym for its starting address. Thus, after an assignment such as

```
pvalues = values;
```

the statements

```
int i;
i = values[0];
```

and

```
int i;
i = *pvalues;
```

are entirely equivalent. Similarly, the statements

```
values[0] = 40;
```

and

```
*pvalues = 40;
```

are exactly equivalent.

The ability to use a pointer instead of referencing element zero of an array would be just a curiosity in C if there were no easy way to use pointers to select other elements of an array. However, C allows you to perform *arithmetic operations* on pointers that change the array element they point to. For example, if **pvalues** points to the beginning of the array **values**, then **pvalues + 2** points to **values[2]**. More generally, any integer expression can be added to a pointer. Similarly, the assignment statement

```
values[2] = 45;
```

is equivalent to the statement

```
*(pvalues+2) = 45;
```

In pointer arithmetic, C automatically increments or decrements the pointer by the proper amount for the type of variable the pointer is declared as pointing to. For example, if **pd** is declared as a pointer to a **double** and **j** is an integer, then **pd + j** points to the address that is **j** * *the number of storage locations needed for a double* beyond the address pointed to by **pd**.

Operations such as adding or subtracting a constant from a pointer are allowed in C. Multiplication, division, or modulo arithmetic on pointers are not allowed. It is legal to subtract one pointer from another. For example, you might have two pointers to elements in an array of **float**s, and the difference between those two pointers minus 1 would be the number of values in the array between the two pointers.

C does not allow you to add two pointers together. Operations on pointers to different data types are also not allowed. For example, subtracting a pointer to a **double** from a pointer to an **int** is illegal.

Pointers of the same type can also be compared. For example, if you had two pointers **p1** and **p2** that pointed to different elements in the same array, you might want to check which pointer was "deeper" into the array with the comparison **p1 < p2**.

While the relationship between pointers and array names in C is very strong, they are not entirely identical. In particular, while the name of an array is an address, its value cannot be changed. Thus, if **pd** is a pointer, the statement

```
pd = pd + 1; /* valid if pd is a pointer */
```

is entirely legal. However, if **values** is an array, the statement

```
values = values + 2; /* not valid if values is an array */
```

is not a legal C statement.

As an example of the uses of pointers, we rewrite the fragment of code that reads values into an array of **float**s. The original code,

```
/* read 1000 values into array x */
float x[1000];
int i;
for(i=0; i<1000; i++)
  scanf("%f", &x[i]);
```

could have been written as follows:

```
/* another way to read 1000 values into x */
float x[1000];
float *px;
for(px = x; px < x + 1000; px++)
  scanf("%f", px);
```

In this example, the pointer **px** is used instead of an array subscript. The condition **px < x+ 1000** uses pointer arithmetic and comparison of two pointer values. The statement **px++** increments **px**, moving it through the original array. Finally, since **px** is a pointer to a **float**, it can be used as an argument to **scanf()** instead of the awkward construction **&x[i]**.

4.2.3 Arrays as Arguments to Functions

C allows you to pass the name of an array as an argument to a function. *However, when an array name is passed as an argument, the value passed to the function is a copy of the address of the beginning of the array, not a complete copy of the entire array.* This is entirely consistent with the interpretation of an array name as a synonym for the address of the zeroth element in the array. Because the array is not copied when its name is used as an argument, the function that is passed the array name is free to interpret that argument either as an array name or a pointer to a value of the same type as the array.

For example, if we wanted to write a general-purpose function that averaged the elements in an array of `float`s, we might code it as follows:

```
/* function to average array of floats */
float average_floats(
  float a[],    /* array of elements to be averaged */
   int n)       /* number of elements to be averaged */
{
  float average = 0.0;
  int i;
  for(i=0; i<n; i++)
    average = average + a[i];
  return(average/n);
}
```

Note that in this example **a**, the argument to the function which is the array of values to be averaged, is declared as

```
float a[];
```

The reason there is no value between the brackets to denote the number of elements in **a** is that the value passed to the function is simply the starting address of the array. The function does not need to know how many elements are in the array because it does not allocate any storage in memory for the array. Only an address is passed to the function. The argument **n** is used to convey how many elements in the array are to be averaged.

This function might be invoked with the following program:

```
/* main function using average_floats() */
#include <stdio.h>
#define SIZE 1000
float average_floats(float[], int);

main()
{
```

```
    float x[SIZE];
    int i;
    for(i=0; i<SIZE; i++)
      scanf("%f", &x[i]);
    printf("Average is %f\n",average_floats(x, SIZE));
  }
```

This program also illustrates a common use of the **#define** preprocessor directive. Defining **SIZE** to be the number of elements in the array makes it easy to change the program for different numbers of values to be averaged. Rather than changing the value in three different places, only the **#define** statement would have to be changed. The program also shows the prototype for an array argument in the declaration of **average_floats()**. The prototype

```
float average_floats(float *, int);
```

would be equivalent.

The function **average_floats()** could also be coded using a pointer to the beginning of the array as an argument and using pointer arithmetic to loop through the array entries. This would be implemented as follows:

```
/* using pointers to average array of floats */
float average_floats(
   float *pa, /* pointer to start of array */
   int n)     /* number of elements to be averaged */
{
  float average = 0.0;
  float *pf;

  for(pf = pa; pf-pa < n; pf++)
    average = average + *pf;
  return(average/n);
}
```

The close relationship between arrays and pointers can also be used to pass portions of large arrays to functions. For example, if you wanted to average elements 10 through 24 of a large array **x** that had 100 elements, you could invoke the function **average_floats()** as

```
float subaverage;
subaverage = average_float( x+10, 15);
```

Whether you use pointers or array notation is largely a matter of taste. For most linear algebra problems, array notation often expresses the intended purpose of a function more clearly than the same function written using pointers. However, as we

shall see in Section 4.4, the use of subscript notation for arrays with multiple dimensions can be somewhat limiting compared to the use of pointers.

4.2.4 Initialization of Arrays

Arrays can be initialized in C as part of their definitions. For example, suppose we wanted to create an array that contained the passing grades on six different quizes. The following C statement both declares the array and initializes its values:

```
float grades[6] = {65, 55, 65, 70, 65, 65};
```

The initial values are enclosed in curly braces. If the array is defined as having more elements than the number of initial values, then the remaining elements are initialized to zero. Thus, the definition

```
float y[2000] = {0.0};
```

initializes all 2000 elements in the array **y** to zero.

You are also allowed to have the size of an array determined automatically by the number of initializing values. For example, the declaration

```
float grades[] = {65, 55, 65, 70, 65, 65};
```

would automatically define the array **grades** to have six elements.

4.2.5 Subscript Checking in C

The C language does not perform any checking when referencing an element of an array. It will therefore accept subscripts that reference places in memory beyond the bounds of the array. This will usually produce serious (and often hard to diagnose) errors. For example, the following is legal C code that will rarely do what any programmer actually intended:

```
double x[1000];
int i;
for(i=0; i<1001; i++)
  scanf("%lf", &x[i]);
```

The advantage of computer languages that, unlike C, check whether subscripted references to arrays are inside the array's defined boundaries is that such languages often detect a common type of programming error. However, array subscript checking adds a significant computational burden to using arrays, slowing down many computational procedures. In addition, there are some situations where a programmer has a good

reason for subscripting outside the memory allocated for an array. The fact that C does not prevent the programmer from subscripting outside the boundaries of an array is entirely consistent with the C's philosophy of providing fast computation and allowing the programmer as much flexibility as possible.

4.3 REPRESENTING STRINGS IN C

C

Arrays of **char**s are used in C to store character strings. For example, the array **str** defined with the statement

```
char str[20];
```

can be used to store a character string of up to 20 characters. Arrays of this type are used for input and output of strings with **scanf()** and **printf()** using the **%s** format in the control string.

For example, the following fragment of code reads two strings of up to 20 characters and then outputs their values along with some commentary.

```
/* fragment using strings in scanf() and printf() */
char str1[20], str2[20];
scanf("%s %s", str1, str2);
printf("The two strings are %s and %s\n", str1, str2);
```

When used in **scanf()**, the **%s** format control will read consecutive characters that are not separated by white space as a single string. Thus, the input

```
Hello world
```

would be treated as two distinct input strings, while the input

```
Hello,world
```

would be treated as a single string.

By convention, the end of a character string in C is denoted by a special marker, called the *null character*, after the last real character in an array. The null character is represented by the special character constant **'\0'**. [1]

For example, the string representing the word **"example"** would be stored in an array of at least eight characters as shown in Figure 4.2. Each element of the array is a **char**. For example, the value of **str[4]** is the character **'p'**.

[1]The reader should not confuse the null character, as represented by the character constant **'\0'**, with the null pointer. The former occupies a single byte, while the latter occupies as many bytes as required by a pointer.

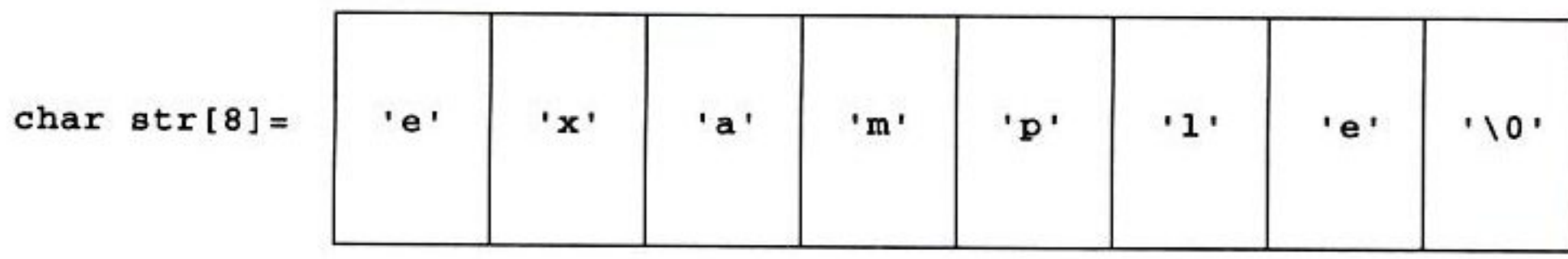


Figure 4.2 Example of using an array of char for a string.

Strings can be initialized in C just as any other array. For example, the variable definition

```
/* initialization of a string */
char str[] = {'e', 'x', 'a', 'm', 'p', 'l', 'e', '\0'};
```

would create an array of eight characters containing the string **"example"** . C also provides a more convenient form in which using double quotation marks (" ") enclosing the initial value. In this form, we would use the definition

```
/* simpler initialization of a string */
char str[] = "example";
```

Note that in the second form, the null character is not explicitly included. This is because the C compiler automatically appends the null character to string constants such as **"example"**.

The standard C library provides a library of functions for manipulating strings. The header file containing the prototype for these functions is **string.h**. In these functions character strings are passed and returned as pointers to the beginning of the array of characters (i.e. with type **char ***). For example, the library function **strcpy()** has two arguments, each of which is a pointer to a character array. It copies the contents of the array of **char** pointed to by the second argument to the similar array pointed to by the first argument. This copies an entire character string, up to and including the null character that terminates the string. **strcpy()** also returns the pointer to the first string. A typical use of **strcpy()** is shown in the following code fragment.

```
/* fragment showing copying of a character string */
#include <string.h>

char str1[]= "test string";
char str2[20];

strcpy(str2, str1);
```

The function call to **strcpy()** would copy the first 12 characters (the four in the word **test**, the blank space between the two words, the six in the word **string**, and the

null character that C always appends to a string constant) from the array **str1** to the array **str2**. The remaining eight characters in the array **str2** would be unchanged.

The most commonly used of the string manipulation functions are as follows:

KEY PARTS OF THE STRING FUNCTION LIBRARY

Function	Arguments	Description of Key String Functions
`strcpy()`	`char *s1, char *s2`	copies **s2** to **s1**; returns **s1**
`strncpy()`	`char *s1, char *s2, int n`	copies at most **n** characters of **s2** to **s1**; returns **s1**
`strcat()`	`char *s1, char *s2`	concatenates **s2** to end of **s1**; returns **s1**
`strncat()`	`char *s1, char *s2, int n`	concatenates first **n** characters of **s2** to end of **s1**; returns **s1**
`strcmp()`	`char *s1, char *s2`	compares **s1** to **s2**; returns **int** < 0 if **s1** < **s2**, 0 if **s1==s2** or > 0 if **s1>s2**
`strlen()`	`char s1`	returns **int** equal to length of **s1** (not including **'\0'**)

The following example illustrates how some of the string manipulation functions are used.

```
/*  test various functions in string library */

#include <stdio.h>
#include <string.h>

main()
{
  char string1[] = "elephant"; /* first test string */
  char string2[] = "bull ";    /* second test string */
  char s3[20], s4[20], s5[20];      /* empty strings */

/* test strlen()*/
  printf("Length of string 1 is %d\n", strlen(string1));

/* test string copy functions */
  strcpy(s3, string2);
  strncpy(s4, string2, 3);
  printf("Two copied strings are %s and %s\n", s3, s4);

/* test string concatenation functions */
  strcat(s3, string1);
  strncat(s4, string1, 3);
  printf("Two concatenated strings are %s and %s\n", s3,
    s4);
```

```
/* test string comparison */
  if(strcmp(s3, s4))
    printf("Strings are different\n");
  else
    printf("Strings are identical\n");
}
```

The simple program initializes two strings and uses the string length, copy, concatenation, and comparison functions. When compiled and executed, it produces the following output:

```
Length of string 1 is 8
Two copied strings are bull  and bul
Two concatenated strings are bull elephant and bulele
Strings are different
```

4.4 VECTOR OPERATIONS

We can use arrays as a convenient way of representing vectors. For example, if **v** is a vector with n entries, then the *length* of **v** (generally denoted as $|\mathbf{v}|$) is the square root of the sum of squared entries in **v**. The following C function takes an array of **double**s representing the vector and the value of n as inputs, and returns the vector's length as a **double**. The code is as follows:

```
/* function to compute the length of a vector */

#include <math.h>
double length(double v[], int n)
{
  double total = 0.0;
  int i;
  for(i=0; i<n; i++)
    total += v[i]*v[i];
  return (sqrt(total));
}
```

The file **math.h** must be included because it contains the prototype for the C math library function **sqrt()** that computes square roots.

Another example is the computation of the *inner product* of two vectors. A graphical interpretation of the inner product is that it finds a scalar value which equals the product of the lengths of the vectors times the cosine of the angle between them. The inner product of two vectors **v** and **w** is usually denoted as $\mathbf{v} \cdot \mathbf{w}$. In C, the following

function computes the inner product of two vectors each with n entries. The vectors are each represented with an array of **doubles**.

```
/* function to compute an inner product */

double inner_product(double v[], double w[],  int n)
{
  double total = 0.0;
  int i;
  for(i=0; i<n; i++)
    total += v[i]*w[i];
  return (total);
}
```

Yet another common problem is finding the *vector product* of two vectors. While the idea of vector products can be generalized to any number of dimensions, the most common case requires the vector product of three-dimensional quantities, with the dimensions representing the x, y, and z axes. The result of a vector product is itself a vector. If **v** and **w** are two vectors, then their vector product is typically denoted by $\mathbf{v} \times \mathbf{w}$.

We need to pass as arguments the two vectors we want to multiply and a vector into which the function will place the result. No return value is needed.

The C function **vector_3_product()** given below has as its arguments **v** and **w**, the two vectors to multiply, and the vector **result** into which the product of **v** and **w** is placed. No return value is needed, so the function is declared as **void**.

```
/* compute vector product for three dimensions */

void vector_3_product(
  double v[], /* first vector */
  double w[], /* second vector */
  double result[]) /* computed product */
{
  result[0] = v[1]*w[2] - v[2]*w[1];
  result[1] = v[2]*w[0] - v[0]*w[2];
  result[2] = v[0]*w[1] - v[1]*w[0];
}
```

A brief example can illustrate the potential value of even this small library of vector functions. Consider a box defined by three vectors **a**, **b**, and **c**, all of which begin at the same point as shown in Figure 4.3. The volume V contained in the "box" (which is technically a parallelepiped) can be shown to be given by the formula

$$V = |(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}|$$

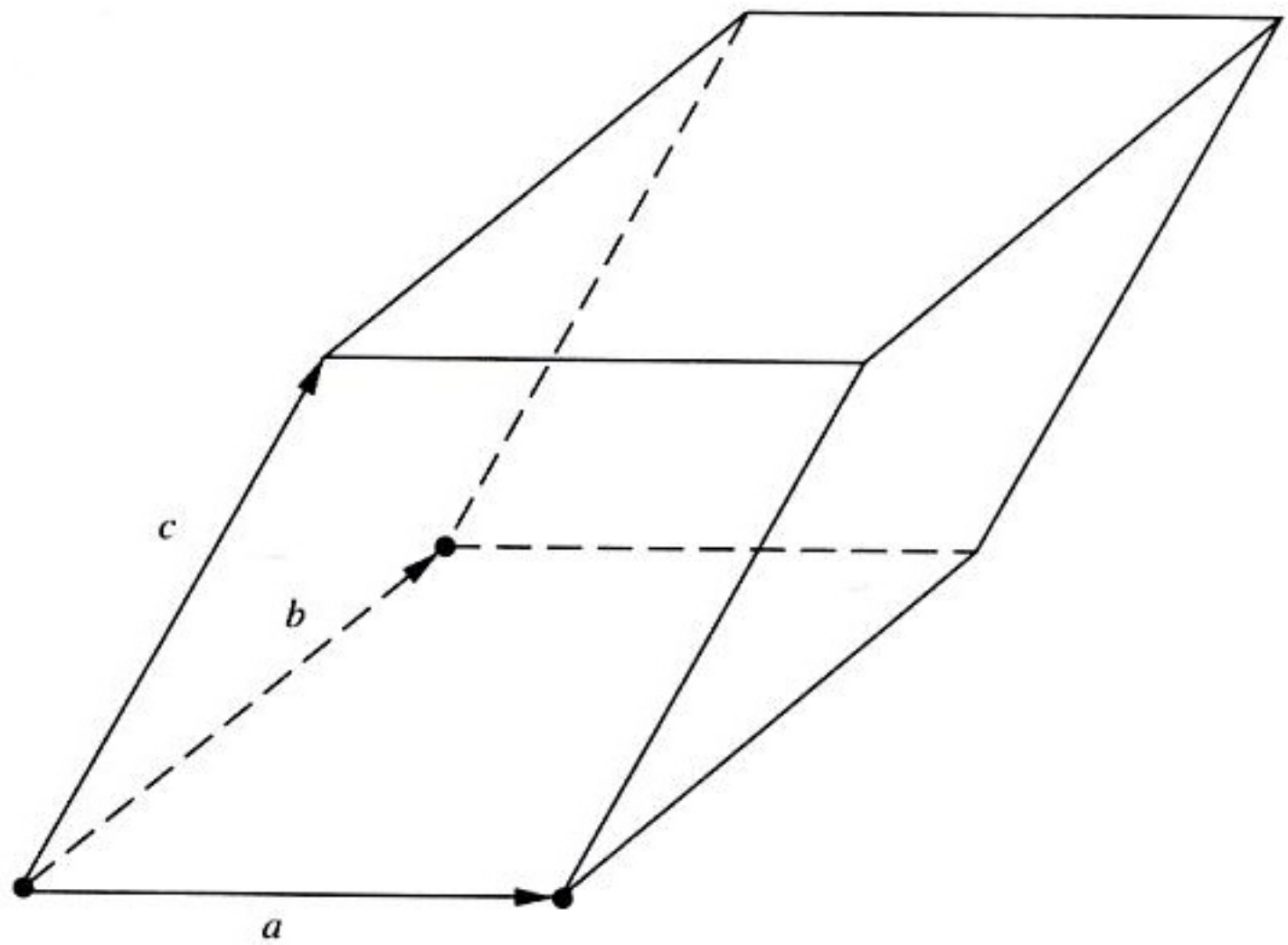


Figure 4.3 Volume defined by three vectors with a common origin.

If we had the three vectors **a**, **b**, and **c** originating at the point (0,0,0) and ending at (1.7,0,0), (0,2.3,0), and (1,2,3), respectively, then the following program could be used to find the volume of the box these vectors define.

```
/* main program to compute volume of parallelepiped */
#include <stdio.h>
#include <math.h>

double inner_product(double *, double *,  int);
double length(double *, int);
void vector_3_product(double *, double *, double *);

main()
{
  double a[] = {1.7, 0.0, 0.0}; /* vector a */
  double b[] = {0.0, 2.3, 0.0}; /* vector b */
  double c[] = {1.0, 2.0, 3.0}; /* vector c */
  double temp[3]; /* temporary vector for a x b */
  double volume;

  vector_3_product(a,b,temp);
  volume = fabs (inner_product(temp, c, 3) );
  printf("Volume is %lf\n", volume);
}
```

4.5 PROGRAMMING PROJECT: POPULATION FORECASTING

In this section we use arrays to build a computer simulation to forecast future growth of a country's population. We will use what is termed a *discrete step* simulation. In this type of model, we update the state of some known system in a series of steps, each corresponding to some fixed time interval.

In the case of population forecasting, the state of the system will be represented by the number of people in each of a set of age and sex categories. For example, the array **`males`** will hold the number of males in the population. Each entry in the array **`males`** will correspond to a range of ages. **`males[0]`** will hold the number of males between the ages of 0 and 9, **`males[1]`** will hold the number of males between ages 10 and 19, etc. We will define **`males`** to have 10 entries, with the last entry, **`males[9]`**, used to hold the number of males 90 years of age or older. The array named **`females`** will be defined similarly to hold the population of females by age group.

We will model population change in 10-year intervals. Over that period, the number of people of either sex in any given age group depends on four factors: mortality, births, aging, and net immigration.

The mortality process will be predicted using age- and gender-specific mortality rates. The arrays **`male_mortality`** and **`female_mortality`** will hold these mortality rates. For example, **`male_mortality[i]`** will contain the fraction of males in age category **`i`** that will die in a 10-year period.

The second process in population dynamics is the birth of infants. It is common to predict births using age-specific birth rates applied to the population of women. This is usually represented in "live births per woman." We shall define the array **`birth_rate`** to store these values, with the value **`birth_rate[i]`** holding the average number of live births in a 10-year period for women in age category **`i`**. For simplicity, we will assume that live births are divided equally between male and female babies.[2]

The aging of the population can be represented straightforwardly. In any time interval, the surviving population of both men and women in age interval i will become the population in age interval $i + 1$. The only exception to this is that men and women in the oldest age group (those 90 or older) stay in their respective age categories.

Finally, there may be some net outmigration or immigration that adds to, or subtracts from, the population in any age group. Net immigration is often the most difficult component of demographic change to predict. This is particularly true for smaller countries that may border regions of political upheaval, where large numbers of refugees may migrate across a border. For the sake of simplicity, we will assume that net in-migration is negligible in this example.

We can express each of these demographics processes in a C function. The first function, named **`mortality()`**, handles the deaths in the population. Its implementation

[2]The assumptions on birth rates are not exactly true. Slightly more than half of all babies born are males. This is generally compensated for by a higher male infant mortality rate. In addition, situations where the number of adult males is decreased, as in post-World War II Soviet Union, may reduce birth rates.

is as follows:

```
/* function to compute deaths */
void mortality(
  double pop[], /* population array */
  double rates[], /* mortality rate */
  int n) /* number of age categories */
{
  int i;
  for(i=0; i<n; i++)
    pop[i] *= (1.0-rates[i]);
}
```

This function has three arguments. The first is an array **pop** holding some population, and the second is the array **rates** that holds the mortality rates, and the third is an **int n**, the number of age categories. The function simply multiplies the population in each age category by one minus the mortality rate for that age category.

The computation of the number of births can be done using a function we have already written. This is because the number of children born is simply the inner product of a vector representing the population of females and the vector of birth rates. For completeness, we repeat the code of the function **inner_product()**:

```
/* function to compute an inner product */

double inner_product(
  double v[],  /* first vector */
  double w[], /* second vector */
  int n) /* number of entries in vectors */
{
  double total = 0.0;
  int i;
  for(i=0; i<n; i++)
    total += v[i]*w[i];
  return (total);
}
```

This function would be called using the arrays **females** and **birth_rate** as the first two arguments, and the number of age categories as the third.

The third function, named **aging()**, represents the aging process. Its implementation is as follows:

```
/* function to age population */
void aging(
  double pop[],          /* population array */
  int n)                 /* number of age categories */
```

```
{
  int i;

  pop[n-1] += pop[n-2];
  for(i=n-2; i > 0; i--)
    pop[i] = pop[i-1];
  pop[0] = 0.0;
}
```

This function has two arguments. The first is an array **pop** holding some population, and the second is an **int n**, the number of age categories. The function first updates the oldest age category by adding the number in the next-to-last age group to the current cohort. It then updates all the other age categories. Finally, it sets the population in the youngest age category to zero.

We will also need functions to read in an array of values and to output an array. The functions **read_vector()** and **output_vector()** below serve this purpose.

```
/* function to read a vector */

void read_vector(
  double array[], /* the array */
  int n)          /* number of values in array */
{
  int i;
  for(i=0; i<n; i++)
    scanf("%lf", array+i);
}

/* function to output a vector */

void output_vector(
  double vec[],       /* vector to be output */
  int n)              /* number of entries in vector */
{
  int i;
  for(i=0; i<n; i++)
    printf("  %d   %10.0lf\n", i, vec[i]);
}
```

Note that the function **output_vector()** displays the subscript index as well as the corresponding value.

The last step is to combine these functions into a single program. Our program begins with the usual **#include** directives and has a series of declarations for the various constants, variables, and function prototypes. The **main()** program then reads in the number of time intervals to be simulated and makes calls to the different functions to perform the simulation. While the functions **mortality()** and **birth()** are coded

to allow for the possibility that we might have different mortality and birth rates in each time period, we have coded the implementation of **main()** under the assumption these factors remain fixed over the interval being forecasted.

The C code implementing **main()** is given below.

```
/*  population forecasting system
    using aging and mortality */

#include <stdio.h>
#define AGE_GROUPS  10

/* function prototypes */

void mortality(double[], double[], int);
double inner_product(double[], double[], int);
void aging(double[], int);
void read_vector(double[], int);
void output_vector(double[], int);

main()
{
/* various arrays used in program */
  double males[AGE_GROUPS], females[AGE_GROUPS];
  double male_mortality[AGE_GROUPS]
  double female_mortality[AGE_GROUPS];
  double birth_rate[AGE_GROUPS];

  int i, periods;
  double newborn;

/* read in number of periods and arrays */
  scanf("%d", &periods);
  read_vector(males, AGE_GROUPS);
  read_vector(females, AGE_GROUPS);
  read_vector(male_mortality, AGE_GROUPS);
  read_vector(female_mortality, AGE_GROUPS);
  read_vector(birth_rate, AGE_GROUPS);

/* loop through all periods */
  for(i=1; i<=periods; i++) {
  /* predict mortality */
    mortality(males, male_mortality, AGE_GROUPS);
    mortality(females, female_mortality, AGE_GROUPS);

  /* predict births */
    newborn = inner_product(birth_rate,
                            females, AGE_GROUPS);
```

```
    /* predict aging */
      aging(males, AGE_GROUPS);
      aging(females, AGE_GROUPS);

    /* update number of children in first age category */
      males[0] = newborn/2.0;
      females[0] = newborn/2.0;

    /* output current populations */
      printf("\nPopulation of males for period %d\n", i);
      output_vector(males, AGE_GROUPS);
      printf("\nPopulation of females for period %d\n",i);
      output_vector(females, AGE_GROUPS);
    }
  }
```

To see how this model works, consider an entirely hypothetical population. The initial populations of males and females and the age-specific mortality rates are given by the following table. Note that this hypothetical population is relatively young. This is characteristic of many third-world agrarian countries where population growth is quite rapid and reductions in infant mortality have not yet been offset by decreases in birth rates.

HYPOTHETICAL POPULATION AND MORTALITY RATES

Age Range	Females	Males	Female Mortality	Male Mortality
0-9	2400	2400	.05	.05
10-19	2300	2300	.05	.05
20-29	1200	1200	.05	.05
30-39	1100	1100	.05	.05
40-49	1100	1000	.05	.07
50-59	800	600	.15	.20
60-69	400	200	.25	.30
70-79	200	100	.35	.40
80-89	170	70	.75	.80
90+	130	30	.75	.80
TOTAL	9800	9000	NA	NA

The model also requires the expected number of live births per woman, by age. We will assume values that are characteristic of still-growing populations as given in the following table:

HYPOTHETICAL BIRTH RATES

Age Range	Birth Rate
0-9	0.0
10-19	.3
20-29	1.2
30-39	1.0
40-49	.2
50-59	0.0
60-69	0.0
70-79	0.0
80-89	0.0
90+	0.0

We can use these assumptions as inputs to the simple demographic model to predict the population of males and females in the future. Figure 4.4 depicts how the

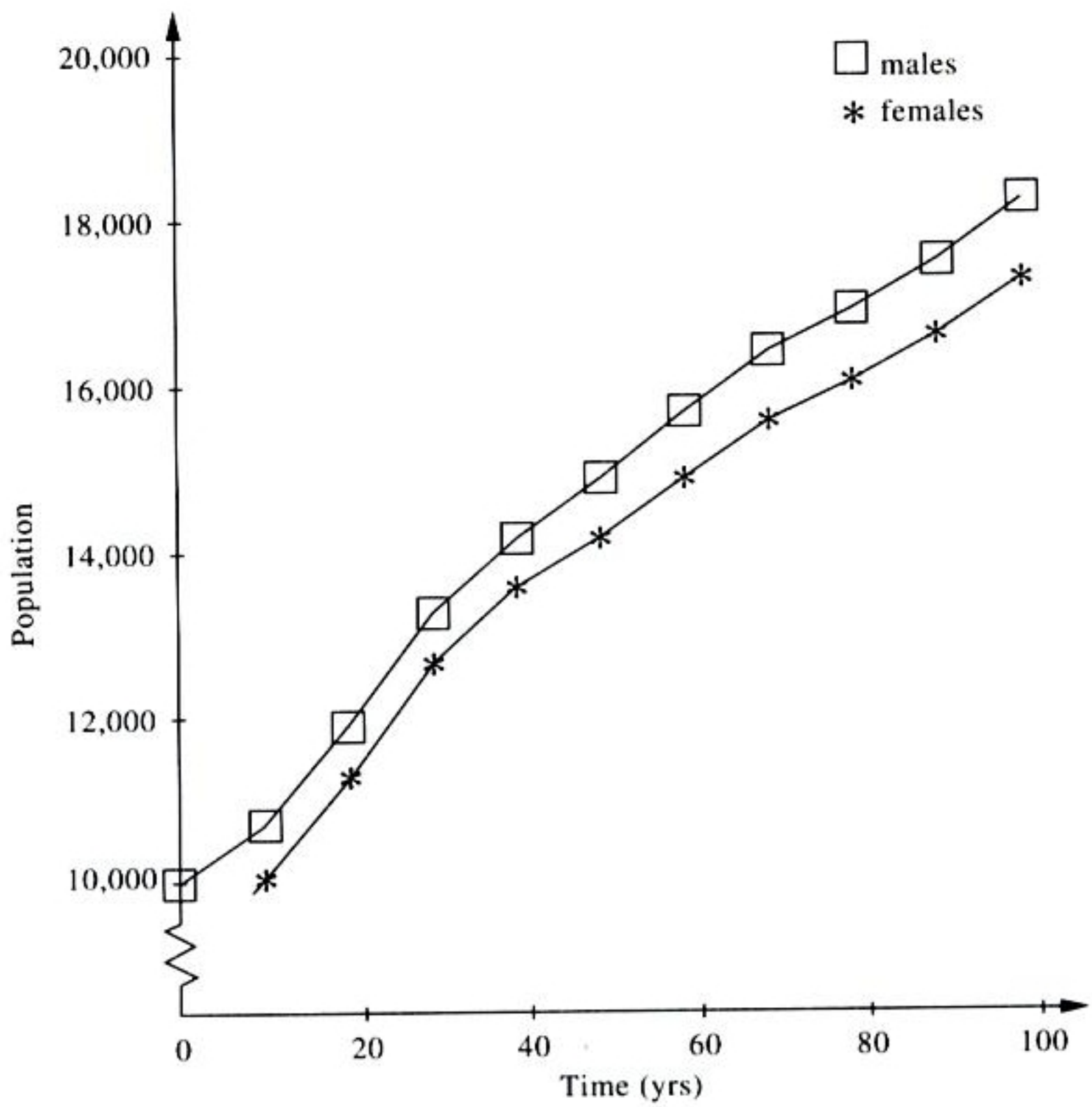


Figure 4.4 Evolution of population over time.

total number of males and females evolve over time. The growth of the population over time in this example is fueled by the combination of a relatively young population and the high birth rate.

4.6 REPRESENTING MATRICES IN C

C

4.6.1 Multidimensional Arrays

The basic idea of arrays in C generalizes to accommodate matrices reasonably straightforwardly. For example, a matrix of **doubles** with three rows and two columns could be defined with the following C statement:

```
double mat[3][2];
```

The two array indices enclosed in square brackets denote the fact that the array is two-dimensional. Higher-order arrays can be defined simply by adding more subscripts.

The rules for referencing elements of a matrix in C are identical to those for single dimensional arrays. As with single-dimensional arrays, the subscripts begin with zero. Thus, the element in the first row and column of the array **mat** defined above would be **mat[0][0]**.

C stores the elements of multidimensional arrays in sequential memory locations. The rule in C is that the elements of a multidimensional array are stored in sequential memory locations with the *outer subscripts varying before the inner ones*. Thus in our example, the elements of **mat** would be stored in the following order:

```
mat[0][0], mat[0][1], mat[1][0], mat[1][1],
           mat[2][0], mat[2][1]
```

This is shown diagramatically in Figure 4.5. In this figure, the six elements of the array **mat** are arranged in sequential memory locations, with the outermost subscript varying first.

As with single-dimension arrays, it is possible to use pointers to address elements within a multidimensional array. For example,

```
mat[i][j]
```

is the same as

```
*(&mat[0][0] + 2*i + j)
```

In the form ***(&mat[0][0] + 2*i + j)**, the expression **&mat[0][0]** evaluates to a pointer to the address where the array **mat** begins. From this beginning address, we can

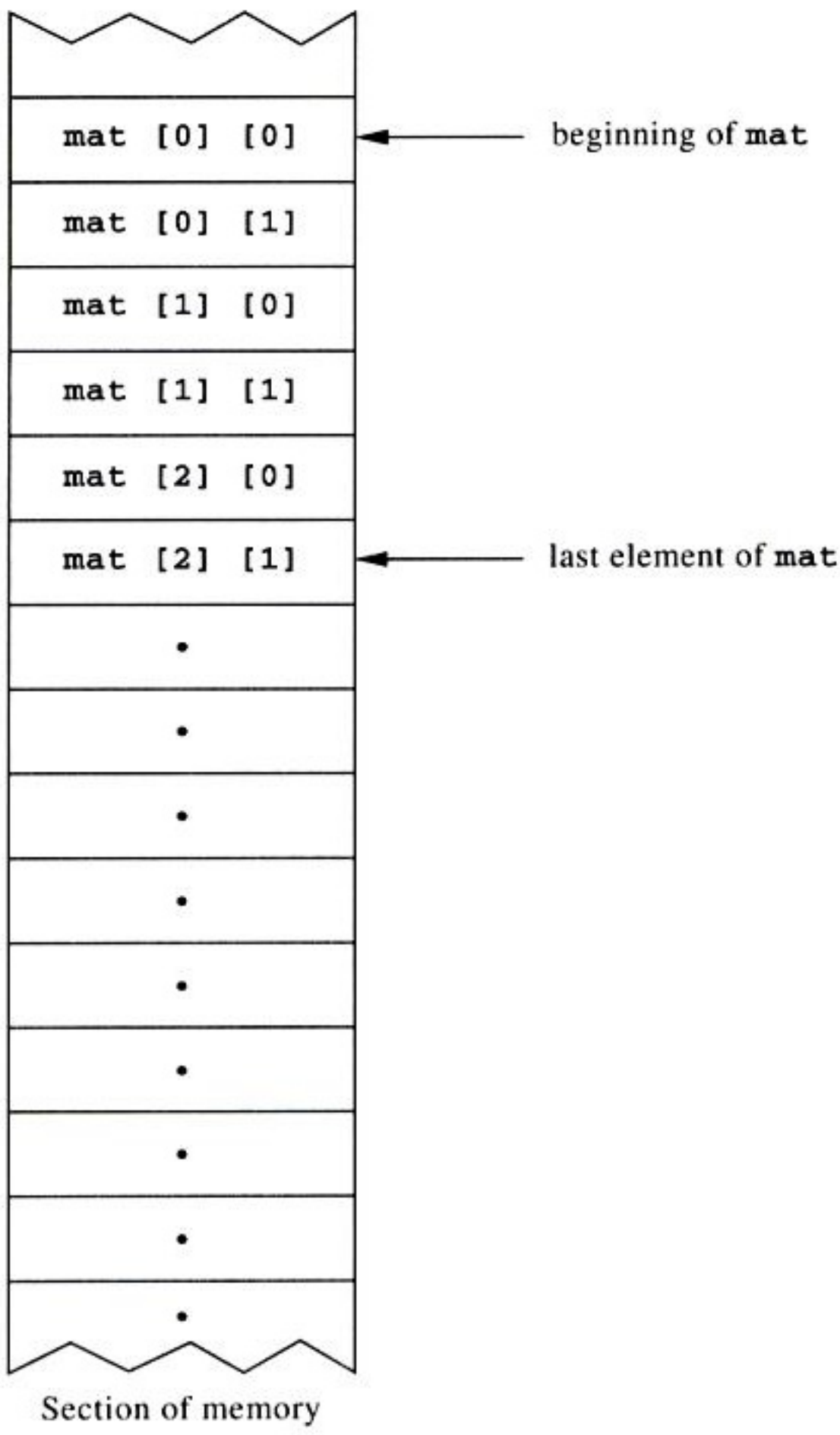


Figure 4.5 Two-dimensional array in memory.

find the number of array entries beyond the first that the element `mat[i][j]` lies. In this case, because `mat` has two columns, the entry `mat[i][j]` lies `2*i+j` elements beyond `mat[0][0]`. Finally, the indirection operator, `*`, outside the parentheses produces the value pointed to by `&mat[0][0]+2*i+j`

The use of a pointer to the starting address of an array (sometimes referred to as a *base address*) incremented by the number of elements beyond the start of the array (sometimes referred to as an *offset*) exploits the properties of pointer arithmetic. This approach works regardless of the type of the array because the rules for pointer arithmetic in C automatically take the storage requirements of the C data types into account. In the example above, the address computed in the pointer arithmetic expression

```
&mat[0][0]+2*i+j
```

will automatically take into account the storage required by `doubles` in C.

More generally, if a two-dimensional array **m** is defined as having **ncol** columns, then the element **m[i][j]** can be equivalently written as ***(&m[0][0]+i*ncol+j)**. While the second form is considerably more awkward than the first, it is often used because of the way multidimensional arrays are passed as arguments to functions. This issue is explored in depth in the next subsection.

4.6.2 Multidimensional Arrays as Arguments to Functions

Multidimensional arrays are passed as arguments to functions in the same way as single-dimensional arrays. Thus, only the address to the beginning of the array is passed. One important difference, however, between single- and multidimensional arrays is that in order for a function to make use of a multidimensional array as an argument, *the function must know all but the innermost dimension of the array*.

For example, consider a function that simply adds up all the elements of **m**, an array of **doubles**, and returns the sum as a **double**. The following code shows such a function.

```
/* function to sum values in a matrix with two columns */

double sum_vals(
  double m[][2], /* the array */
  int nrows)     /* number of rows */
{
  int i,j;
  double sum = 0.0;
  for(i=0; i<nrows; i++)
    for(j=0; j<2; j++)
      sum += m[i][j];
  return(sum);
}
```

The reason that the function **sum_vals()** needs to know the number of columns in the array is that without such information it cannot compute how many elements beyond the beginning of the array the element **m[i][j]** lies. Put another way, the reference to **m[i][j]** is translated into ***(&m[0][0]+i*NCOLS+j)**, where **NCOLS** is number of columns in **m**. Unless **NCOLS** is known, the C compiler cannot compute the appropriate quantity to add to **m** to obtain the value in the ith row and jth column.

As coded above, the function **sum_vals()** is extremely limited in its usefulness. It works only on arrays that have two columns rather than on all arrays. For this reason, a more general-purpose function that adds the elements in an array would usually be coded using a pointer to the first element in the array along with the number of rows and columns in the array as its arguments. It would then use pointer arithmetic to compute the *offset* from the beginning of the array to access the array's elements. One possible

implementation is as follows:

```
/* general function to sum the values in a matrix */

double sum_vals(
  double *m,   /* pointer to first element of array */
  int nrows,   /* number of rows */
  int ncols)   /* number of columns */
{
  int i,j;
  double sum = 0.0;
  for(i=0; i<nrows; i++)
    for(j=0; j<ncols; j++)
      sum += *(m + i*ncols + j);
  return(sum);
}
```

If the array **mat** has three rows and two columns, this function would be invoked as in the following code fragment:

```
double mat[3][2], sum;

sum = sum_val(&mat[0][0], 3, 2);
```

Note that the first argument is **&mat[0][0]**, which is the address of the first element of the array. This is technically different from the variable **mat**, which is pointer to a two-dimensional array.

To understand the distinction between a pointer to the first element of a two-dimensional array and a pointer to the array itself, one must understand how arrays are treated in C. The appropriate way to think about a two-dimensional array is as an *array of one-dimensional arrays*. Thus, if **mat** is a 3 by 2 array of **doubles**, then **mat[0]** and **mat[1]** are both arrays of two **doubles**. Thus the expression

```
*(mat[1] + 1)
```

is entirely equivalent to

```
mat[1][1]
```

The reason this is true is that **mat[1]** is itself an array, and its value is therefore the address of the beginning of that array. When we add **1** to **mat[1]**, we are doing pointer arithmetic. The dereferencing operator (*****) gives the value pointed to by **mat[1]+1**.

Because C treats objects such as **mat[1]** as arrays, the value of **mat** (without any brackets) is the address of **mat[0]**, and its type is a pointer to an array of arrays

of **double**s. The expression **mat+1** increments the pointer **mat** by one array (*not* one **double**), making it a pointer to row 1 of the array **mat**. Thus ***(mat+1)** points to element 0 in row 1, and the values ***(mat+1)[0]** and ***(mat+1)[1]** are legal references to **mat[1][0]** and **mat[1][1]**, respectively.

Some of these complicated relationships are shown in Figure 4.6, which depicts a section of memory where the six elements of the two-dimensional array **mat** are stored. The elements **mat[0]**, **mat[1]**, and **mat[2]** are themselves arrays and are therefore pointers to the place in memory where the arrays begin. Each of those arrays corresponds to a row of the complete array **mat**. In addition, the figure shows the value **mat** as itself an array consisting of **mat[0]**, **mat[1]**, and **mat[2]**. The value of **mat** is a pointer to **mat[0]**, the first element in that array of arrays.

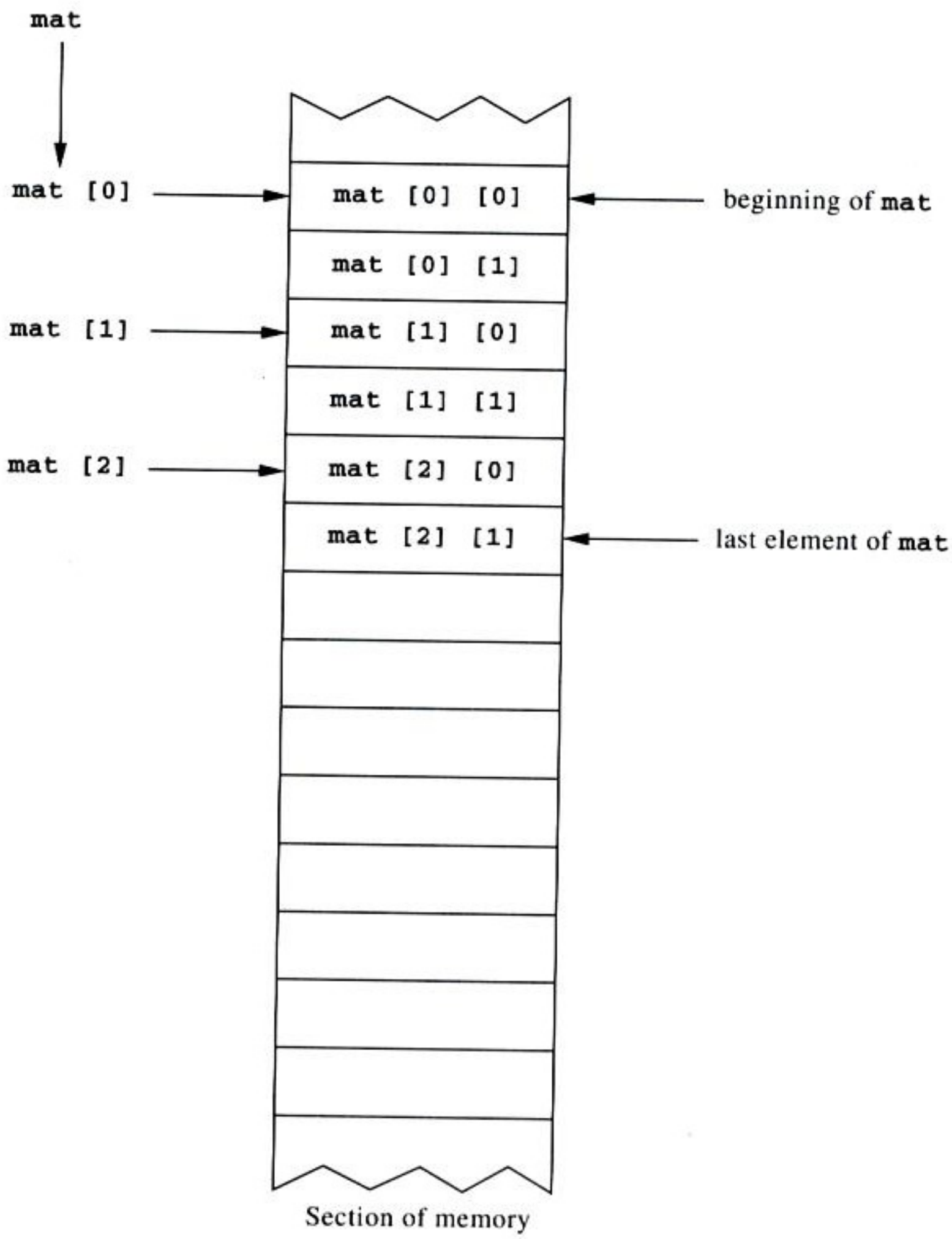


Figure 4.6 Relationships among parts of a two-dimensional array.

One of the consequences of the relationship between pointers and arrays is that there are many equivalent ways of accessing elements of two-dimensional arrays. Consider, for example, an array named **test** defined as

```
int test[10][20];
```

All of the following are equivalent ways of accessing **test[i][j]**:

```
*(test[i]+j)
(*(test+i))[j]
*((*(test+i))+j)
*(&test[0][0] + 20*i + j)
```

4.6.3 Initialization of Multidimensional Arrays

Multidimensional arrays can be initialized in a way that is similar to the method used for one-dimensional arrays. The most commonly used form relies on nested curly braces, as in the definition

```
int vals[3][2] = { {10, 20}, {20, 40}, {5, 10} };
```

You may also list the values without grouping them, as in the statement

```
int vals[3][2] = {10, 20, 20, 40, 5, 10};
```

You may also omit the size of the innermost dimension, so that the statement

```
int vals[][2] = { {10, 20}, {20, 40}, {5, 10}};
```

is legal. All dimensions other than the innermost must be given. The compiler will automatically compute the innermost dimension from the number of initial values provided.

As was the case with one-dimensional arrays, if the array contains more elements than there are initializers, then the remaining values are initialized to zero.

4.6.4 Using Macros with Arguments to Reference Array Elements

Using a base address and computing the offsets makes using multidimensional arrays in functions somewhat awkward. This shortcoming of C can be partly ameliorated by taking advantage of the C preprocessor's ability to have **#define** directives with arguments. This is an extended version of the **#define** directive that was first introduced in Section 2.10. **#define** directives that have arguments are sometimes called *macros*.

Consider, for example, a macro named **CUBE** created by the following **#define** directive:

```
#define CUBE(X) ((X)*(X)*(X))
```

After the macro is defined, an occurrence of the text **CUBE** followed immediately by an argument in parentheses will be replaced by the preprocessor. This replacement is called *expanding* the macro. For example, suppose that the source code file has the statement

```
double a=5.0;
a = CUBE(a);
```

The C preprocessor would expand the macro **CUBE(a)**, changing the statement to

```
double a=5.0;
a = (a)*(a)*(a);
```

It might appear that enclosing the body of the macro and each of the parts of the macro expression in parentheses is unnecessary. However, suppose that the **#define** statement for **CUBE** been written as

```
#define CUBE(X) X*X*X
```

If we used the macro in the statement in the code fragment

```
double a=5.0;
a = CUBE(a+1.0);
```

the preprocessor would transform the assignment statement to

```
double a=5.0;
a = a+1.0 * a+1.0 * a+1.0;
```

which, given the rules for precedence of arithmetic operators, would result in a value of **a** equal to 16 rather than the cube of **a+1.0**, which equals 216.

Macros with arguments look like functions when used in a program. It is important to remember, however, that a macro is simply a rule for text substitution. The arguments of a macro do not have any variable types associated with them, and there is no way to enforce type restrictions in their use. In some situations, macros and functions that appear to do exactly the same thing can have entirely different effects. For example, consider a function named **cube** that returns the cube of an **int**.

```
/* function that returns cube of an int */
int cube(int x)
{
```

```
    return(x*x*x);
}
```

If we use this function in the fragment

```
int i=3;
int j;

j = cube(i++);
```

the result will be that **i** has the value 4 and **j** has the value 27. If we performed the seemingly same operation using the macro **CUBE**, as in

```
int i=3;
int j;

j = CUBE(i++);
```

the preprocessor would expand the macro **CUBE** so that the assignment statement would be

```
j=(i++)*(i++)*(i++);
```

This would result in **j** having the value 20 (**3*4*5**) and **i** having the value 6.

Because macros are fundamentally different from functions, we will always use capital letters for the names of macros. This is a common, though not universal, convention among C programmers.

One use for macros with arguments is as a means of avoiding the awkwardness associated with arrays that are arguments to functions. As an example, we can rewrite the function **sum_vals()** in Section 4.6.2, which sums all the values in a two-dimensional array, as follows:

```
/* function using macros to sum values in a matrix */

/* macro for m[i][j] */
#define M(I,J) (*(m+(I)*ncols+(J)))

double sum_vals(
  double *m,   /* pointer to first element in array */
  int nrows,   /* number of rows */
  int ncols)   /* number of columns */
{
  int i,j;
  double sum = 0.0;
```

```
    for(i=0; i<nrows; i++)
    for(j=0; j<ncols; j++)
      sum += M(i,j);
  return(sum);
}

#undef M
```

The macro **M** has two arguments, **I** and **J**, because references to the array **m** will require a row and column number. The parentheses around the body of the macro, **I** and **J**, avoid some of the problems arising from the fact that macros are only text substitutions, not functions.

The macro gets used in the statement

```
sum += M(i,j);
```

which the C preprocessor converts to

```
sum += (*(m+(i)*ncols+(j)));
```

Note that we introduce a new preprocessor directive, **#undef**. This has the effect of *undefining* a macro so that all statements after the appearance of the **#undef** will not have that macro substitution done. In this case the directive **#undef M** undefines the macro **M**. While this step is not essential, it is good programming practice to undefine a macro after the portion of the code that uses it. This limits the range of statements in your program where the macro is defined, avoiding mistakes if you use the same macro name for a different purpose later.

This new version of **sum_vals()** is used in exactly the same way as the earlier implementation. For example, if we had defined a two-dimensional array named **mat** with three rows, we could use this new version of **sum_vals()** with the statement

```
printf("Sum of values is %lf\n",
       sum_vals(&mat[0][0],3,2));
```

We will use macros for accessing elements of multidimensional arrays in the remainder of this chapter because they make code easier to follow.

4.7 BASIC MATRIX OPERATIONS

We can illustrate the use of multidimensional arrays in C by developing some general-purpose functions for matrix manipulation. For example, in linear algebra, the product of a matrix **A** that has m rows and n columns with a vector that has n elements is a vector with m elements. If the matrix **A** is defined as having elements

```
void mat_vec_mult(
  double *a,  /* pointer to first element in matrix  */
  double b[], /*  vector */
  double c[], /* result of  A*b */
  int m,      /* number of rows in A */
  int n)      /* number of columns in A */
{
  double sum=0.0;
  int i,j;
  for(i=0; i<m; i++) {
      for(j=0; j<n; j++)
        sum = sum + A(i,j) *  b[j];
      c[i] = sum;
    }
}

#undef A
```

The implementation uses a macro named **A** to access elements of the matrix.

To use the function **mat_vec_mult()**, you must pass the first element of the array **a**. For example, the following program multiplies an array with four rows and three columns by a vector with three elements, producing a vector with four elements.

```
/* use matrix times vector function */
void mat_vec_mult(double *,double *,double *,int,int);

main()
{

double mat[4][3] = {{3,2,1}, {5,6,4}, {8,4,2}, {2,1,2}};
double vec[3] = {1,2,3};
double result[4];

mat_vec_mult(&mat[0][0], vec, result, 4, 3);
}
```

The four elements in the array **result** will be 10, 29, 22, 10.

It is worth noting that the product of a matrix and a vector is *not* the same as the product of a vector and a matrix. More generally, operations involving matrix multiplication are not commutative.

Another example of matrix algebra is the multiplication of two matrices **A** and **B**, where **A** has m rows and n columns, and **B** has n rows and p columns. The product of **A** and **B** is a matrix which we shall call **D** that has m rows and p columns. The elements d_{ij} of **D** are given by the following formula:

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

For example, if we have the matrices **A** and **B** as

$$\mathbf{A} = \begin{pmatrix} 3 & 2 \\ 4 & 1 \\ 7 & 9 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 2 & 3 \\ -1 & 4 \end{pmatrix}$$

then the product **AB** is

$$\mathbf{AB} = \begin{pmatrix} 4 & 17 \\ 7 & 16 \\ 5 & 57 \end{pmatrix}$$

The function **mat_mult()** (for matrix multiply) below computes the product of two matrices, placing the result in a third matrix. As with the function **mat_vec_mult()**, the user of the function must make sure that appropriately dimensioned arrays are provided as arguments.

```
/* function to compute product of two matrices */

#define A(I,J) (*(a + (I)*n + (J)))
#define B(I,J) (*(b + (I)*p + (J)))
#define C(I,J) (*(c + (I)*p + (J)))

void mat_mult(
  double *a, /* pointer to first element of A */
  double *b, /* pointer to first element of B */
  double *c, /* pointer to first element of A*B */
  int m,     /* number of rows in A */
  int n,     /* number of cols. in A & rows in B */
  int p)     /* number of cols. in B */

{
  double sum;
  int i,j,k;
  for(i=0; i<m; i++)
    for(j=0; j<p; j++){
      sum = 0.0;
      for(k=0; k<n; k++)
        sum +=  A(i,k) * B(k,j);
      C(i,j) = sum;
      }
}

#undef A
#undef B
#undef C
```

Other common matrix operations include:

- Transposition—constructing a matrix in which element ij replaces element ji.
- Addition—adding two matrices with the same number of rows and columns, with each element computed as the sum of the corresponding elements in the two original matrices.
- Operations on matrices that have a particular form—some matrices have a particular form that makes it possible to perform certain operations more efficiently. For example, a symmetric matrix is one where element ij equals element ji for all i and j. The transposition of a square symmetric matrix is trivial because transposition does not change the matrix at all.

Some specific cases of specialized matrix operations are given as exercises at the end of this chapter.

4.8 SIMULTANEOUS EQUATIONS

Systems of linear equations arise in virtually every area of engineering and science. In addition, many mathematical models in the social sciences produce systems of simultaneous linear equations. These systems are often quite large and can only be solved in reasonable amounts of time with a computer.

We consider here only the case where the number of linear equations is the same as the number of unknowns. We define the unknowns as $x_0, x_1, \ldots, x_{n-1}$. Suppose that we have n equations with n unknowns. The jth equation of the system is of the form

$$a_{j0}x_0 + a_{j1}x_1 + \ldots + a_{j,n-1}x_{n-1} = b_j$$

Systems of linear equations are generally expressed in matrix form. Define a matrix $\mathbf{A}$ of coefficients in the form

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ a_{20} & a_{21} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

and define two vectors, $\mathbf{x}$ and $\mathbf{b}$, with elements $x_0, x_1, \ldots, x_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$, respectively, then a system of linear equations can be written in the matrix form

$$\mathbf{Ax} = \mathbf{b}$$

4.8.1 Simple Gauss-Jordan Elimination

The basic method we will use to solve a system of linear equations is called *Gauss-Jordan elimination*. The method is based on two observations:

- Multiplying any equation in the system by a constant other than zero does not affect the solution. Since each row of the matrix $\mathbf{A}$ represents the coefficients