```
/* File: matdef.h */
#define MAX 10

/* File: matutil.h */
#include "matdef.h"
int readmatrix(double x[][MAX], int r, int c);
void printmatrix(double x[][MAX], int r, int c);
int readvector(double x[], int n);
void printvector(double x[], int n);

/*   File: matutil.c */
#include <stdio.h>
#include "matdef.h"
#include "matutil.h"
/*   Reads a matrix x with r rows and c columns. MAX
     provides the maximum column range for the array.
*/
int readmatrix(double x[][MAX], int r, int c)
{    int i, j;
     double z, sum = 0;

     printf("Matrix data entry:\n");
     for (i = 0; i < r; i++) {          /* for each row of matrix */
          printf("Type a row of %d numbers\n", c);

          for(j = 0; j < c; j++){       /* read c elements of the row */
               scanf("%lf", &z);
               x[i][j] = z;
               sum += z;
          }

     }
     return sum;
}

/*   Prints a matrix with r rows and c columns */
void printmatrix(double x[][MAX], int r, int c)
{    int i, j;

     printf("Matrix is:\n");
     for (i = 0; i < r; i++) {     /* for each row */

          for(j = 0; j < c; j++)   /* print the row */
               printf("%f  ",x[i][j]);

          printf("\n");
     }
}
```

```
/*   Reads a vector of size n. Function returns the sum
     of input values.
*/
int readvector(double x[], int n)
{    int i;
     double sum = 0;

     printf("Type %d numbers, <all zeros to quit>: ", n);
     for (i = 0; i < n; i++) {
          scanf("%lf", x + i);
          sum += x[i];
     }

     return sum;
}

/*   Prints a vector of size n. */
void printvector(double x[], int n)
{    int i;

     printf("Vector is:\n");

     for (i = 0; i < n; i++)
          printf("%f\n", x[i]);
}
```

Figure 15.1: Matrix and Vector I/O Functions

```
/*   File: matutil.h - continued */
void mapvector(double a[][MAX], double x[], double y[],
               int r, int c);


/*   File: matutil.c - continued */
/*   Computes a * x ====> y, where a[][] has r rows and c columns. */
void mapvector(double a[][MAX], double x[], double y[],
                    int r, int c)
{    int i, j;

     for (i = 0; i < r; i++){
          y[i] = 0;

          for (j = 0; j < c; j++)
               y[i] += a[i][j] * x[j];
     }
}
```

Figure 15.2: Code for `mapvector()`

by `mapvector()` into a new vector which is printed. The function `getrc()` shown in Figure 15.4 and is included in file `matutil.c`. The source files `mat.c` and `matutil.c` are compiled and linked and tested producing the following sample session:

```
***Matrices and Vector Transformations***

Rows:  2
Columns:  3

Matrix data entry:
Type a row of 3 numbers
1 2 3
Type a row of 3 numbers
4 5 6

Matrix is:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000

Type 3 numbers, <all zeros to quit>:  1 2 3
Transformed Vector is:
14.000000
32.000000
```

```
/*   File: mat.c
     Other Source Files: matutil.c
     Header Files: matutil.h
     This program reads a matrix. It then repeatedly reads vectors.
     Each vector is transformed by the matrix and printed out.
*/

#include <stdio.h>
#include "matdef.h"
#include "matutil.h"

main()
{    double a[MAX][MAX];
     double x[MAX], y[MAX];
     int r, c;

     printf("***Matrices and Vector Transformations***\n\n");
     getrc(&r, &c);
     readmatrix(a, r, c);
     printmatrix(a, r, c);

     while (readvector(x, c)) {
          mapvector(a, x, y, r, c);
          printf("Transformed "); /* Prefix to printvector() mesg */
          printvector(y, r);
     }
}
```

Figure 15.3: Driver to read and transform vectors

```
/*   File: matutil.h - continued */
void getrc(int * rp, int * cp);


/*   File: matutil.c - continued */
/*   Gets the number of rows and columns for a matrix. rp point to
     rows and cp points to columns.
*/
void getrc(int * rp, int * cp)
{
     printf("Rows: ");
     scanf("%d", rp);
     printf("Columns: ");
     scanf("%d", cp);
}
```

Figure 15.4: Code for `getrc()`

```
Type 3 numbers, <all zeros to quit>:  3 2 1
Transformed Vector is:
10.000000
28.000000

Type 3 numbers, <all zeros to quit>:  2.5 3 4.5
Transformed Vector is:
22.000000
52.000000

Type 3 numbers, <all zeros to quit>:  0 0 0
```

15.1.2 Matrix Operations: Sums and Products

Other common manipulations involving matrices require addition of two matrices, multiplication of two matrices, and inversion of matrices. In this section, we will implement matrix addition and matrix multiplication algorithms. Addition of two matrices may arise when two sets of equations relate the same set of variables. For example, consider the matrix equations:

$$\begin{aligned} A \times X &= Y1 \\ B \times X &= Y2 \end{aligned}$$

Corresponding equations of the two sets can be added together to obtain a combined single set:

$$C \times X = Y$$

```
/*   File: matutil.h - continued */
void matsum(double c[][MAX], double a[][MAX],
               double b[][MAX], int rows, int cols);


/*   File: matutil.c - continued */
/*   Adds matrix a to matrix b to generate a matrix c. Parameters
     r and c specify the rows and columns.
*/
void matsum(double c[][MAX], double a[][MAX],
               double b[][MAX], int rows, int cols)
{    int i, j;

     for (i = 0; i < rows; i++)
          for (j = 0; j < cols; j++)
               c[i][j] = a[i][j] + b[i][j];
}
```

Figure 15.5: Code to add rectangular matrices

where, in matrix terms,

$$\begin{aligned} C &= A + B \\ &\textit{and} \\ Y &= Y1 + Y2 \end{aligned}$$

Vectors are special cases of rectangular matrices having n rows and 1 column. We will therefore implement a single function that sums two rectangular matrices. The sum of matrices A and B generates a new matrix, say C. If the elements of matrix A are `a[i][j]`, and those of B are `b[i][j]`, then the sum matrix, C with elements `c[i][j]`, is determined as follows:

```
c[i][j] = a[i][j] + b[i][j]
```

The implementation of matrix addition is easy, and the code is shown in Figure 15.5.

Multiplication of two matrices A and B results when combining two transformations, i.e. where a vector being transformed by a matrix A is itself the result of a transformation by a matrix B. Consider the following two sets of equations:

$$\begin{aligned} A \times Z &= Y \\ &\textit{and} \\ B \times X &= Z \end{aligned}$$

Since, by the second equation, Z equals $B \times X$, we can substitute $B \times X$ for Z in the first equation:

$$A \times B \times X = Y$$

Or, the combined equation results in:

$$C \times X = Y$$

The product of matrices A and B generates a matrix C. If the number of rows and columns of A are given by $r1$ and $c1$, and those of B are given by $r2$ and $c2$, then, the number of elements of Z represents the number of columns of A and the number of rows of B, i.e. $c1 = r2$. Also, C must have the same number of rows as A and the same number of columns as B. That is, the number of rows and columns of C must be $r1$ and $c2$. It turns out that each `c[i][j]` is a result of a scalar product of row i of matrix A and column j of matrix B. Let the i^{th} row of A and the j^{th} column of B be:

```
a[i][0]  a[i][1]  ...  a[i][c1 - 1]
b[0][j]  b[1][j]  ...  b[r2 - 1][j]
```

then, the scalar product, `c[i][j]`, is given by:

```
a[i][0] * b[0][j] + a[i][1] * b[1][j] + ... + a[i][c1-1] * b[r2-1][j]
```

With this algorithm, the sum is easily implemented as a cumulative sum, initialized to zero. Each pass through the loop adds one product term, `a[i][k] * b[k][j]`, for k from zero through $c1-1$. That is, the following loop computes `c[i][j]`:

```
c[i][j] = 0;
for (k = 0; k < c1; k++)
     c[i][j] += a[i][k] * b[k][j];
```

Such a loop is repeated for all appropriate i rows and j columns. The code for matrix product is shown in Figure 15.6.

We can now write a simple example that uses the matrix functions defined above as shown in Figure 15.7. The program adds and multiplies matrices. To keep the program simple, we assume square matrices. The program first reads in the size of square matrices, and then reads in the two matrices. These matrices are added and multiplied, and the resultant matrices are printed. A sample session is shown below:

```
***Square Matrices - Sums and Products***

Size of square matrices:  2

Matrix data entry:
Type a row of 2 numbers
2 3
Type a row of 2 numbers
3 4
```

```
/*   File: matutil.h - continued */
void matprod(double c[][MAX], double a[][MAX], double b[][MAX],
               int r1, int c1, int r2, int c2);

/*   File: matutil.c - continued */
/*   Matrix multiplication of matrix a (r1 rows and c1 columns)
     and matrix b (r2 rows and c2 columns). Result is matrix c with
     r1 rows and c2 columns.
*/

void matprod(double c[][MAX], double a[][MAX], double b[][MAX],
               int r1, int c1, int r2, int c2)
{    int i, j, k;

     if (c1 != r2) {
          printf("Error - Columns of matrix A do not match rows of B\n");
          return;
     }

     for (i = 0; i < r1; i++)
          for (j = 0; j < c2; j++) {
               c[i][j] = 0;

               for (k = 0; k < c1; k++)
                    c[i][j] += a[i][k] * b[k][j];
          }
}
```

Figure 15.6: Code for matrix product

```
/*   File: matops.c
     Other Source Files: matutil.c
     Header Files: matutil.h
     This  program adds and multiplies two square matrices.
     The matrices are read into two dimensional arrays.
*/
#include <stdio.h>
#include "matdef.h"
#include "matutil.h"
main()
{    double a[MAX][MAX], b[MAX][MAX], c[MAX][MAX];
     int n;

     printf("***Square Matrices - Sums and Products***\n\n");
     printf("Size of square matrices: ");
     scanf("%d", &n);

     readmatrix(a, n, n);
     readmatrix(b, n, n);
     matsum(c, a, b, n, n);

     printf("Sum ");          /* Prefix to msg in printmatrix() */
     printmatrix(c, n, n);
     matprod(c, a, b, n, n, n, n);

     printf("Product ");      /* Prefix to msg in printmatrix() */
     printmatrix(c, n, n);
}
```

Figure 15.7: Driver to test matrix operations

```
Matrix data entry:
Type a row of 2 numbers
4 5
Type a row of 2 numbers
6 7

Sum Matrix is:
6.000000 8.000000
9.000000 11.000000

Product Matrix is:
26.000000 31.000000
36.000000 43.000000
```

Another important matrix operation is the inversion of a square matrix. An inverse matrix has the property:

$$A^{-1} \times A = A^{-1} \times A = I$$

where A^{-1} is the *inverse matrix* and I is a *unit matrix* with unit diagonal elements and zero elements elsewhere. The unit matrix has the property:

$$I \times X = X \times I = X$$

If $A \times X = Y$, it follows that

$$\begin{aligned} A^{-1} \times A \times X &= A^{-1} \times Y \\ & \textit{or} \\ X &= A^{-1} \times Y \end{aligned}$$

Thus, given the inverse matrix, the solution to the matrix equation for any Y is easily obtained.

Inversion of a matrix is somewhat more complex. An inverse of a matrix can be obtained by the Gauss-Jordan method — a modified version of the Gauss elimination method discussed in Chapter 9. A good reference [1], for matrix computational methods as well as other numeric methods, is given at the end of this chapter.

15.2 Complex Numbers

Complex numbers are encountered in many mathematical applications. In this section, we will first review complex numbers and operations involving complex numbers. We will then represent complex numbers using structure types and implement many of the basic complex number operations.

The squares of a real number, either positive or negative, is a positive number. Numbers whose squares are negative cannot be real numbers; they are, therefore, called *imaginary numbers*. Thus,

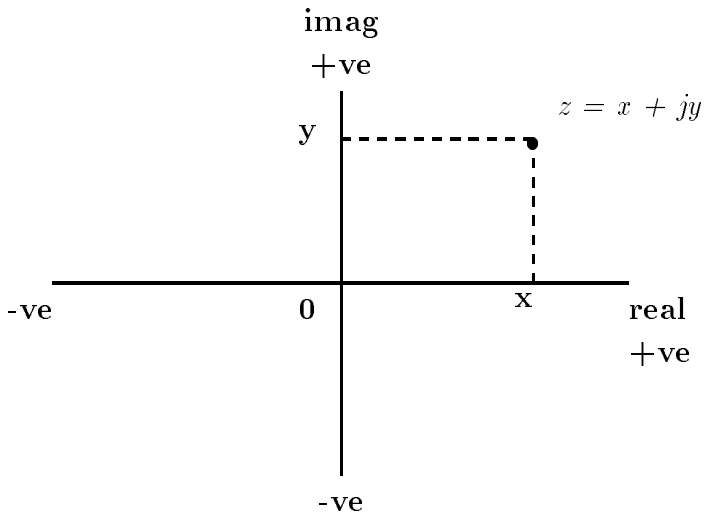

Figure 15.8: Complex Numbers in Rectangular Coordinates

imaginary numbers are the square roots of negative numbers. For example, consider:

$$z = \sqrt{-|x|}$$

Here $|x|$ is the absolute value of x, and thus z is a square root of a negative number, i.e. an imaginary number. Imaginary numbers are written in a normalized manner as follows:

$$\begin{aligned} z &= \sqrt{-1 \cdot |x|} \\ &= \sqrt{-1} \cdot \sqrt{|x|} \\ &= j \cdot y \end{aligned}$$

where, $j = \sqrt{-1}$, and $y = \sqrt{|x|}$. Square root of -1 is represented by the special symbol, i in mathematics or j in Electrical Engineering. Thus, an imaginary number is represented by j times a real number y. A *complex number* is a number that is sums of both a real and an imaginary number.

$$z = x + j \cdot y$$

Both x and y are real numbers, and z is a complex number. Either of the real numbers x or y can, of course, be zero; in which case, the complex number reduces to either a real or an imaginary number. The number x is called the *real part* of z, and y is called the *imaginary part*. Remember that both the real part, x, and the imaginary part, y, are real numbers. It is j that is an imaginary number, not y.

Complex numbers can be visualized geometrically as points on a two dimensional plane with rectangular axes, **real** and **imag**. Then, the real part of a number is the projection of the point onto the real axis, and the imaginary part of the number is the projection onto the imaginary axis imag (see Figure 15.8. For these reasons, the complex number representation as a sum of real and imaginary parts is called *representation in rectangular coordinates.*

Addition, subtraction, multiplication, and division operators for complex numbers are defined in terms of the same operator symbols as for real numbers, viz. `+, -, *, /`. The sum of two

complex numbers is simply the sum of their real parts plus j times the sum of their imaginary parts. Thus, if

$$\begin{aligned} z1 &= x1 + j \cdot y1 \\ z2 &= x2 + j \cdot y2 \end{aligned}$$

then the sum of `z1` and `z2` is given by:

$$z1 + z2 = (x1 + x2) + j \cdot (y1 + y2)$$

The product of `z1` and `z2` is obtained by multiplying the two numbers, replacing $j \cdot j$ by -1, and collecting the real terms and the imaginary terms. Thus,

$$\begin{aligned} z1 * z2 &= (x1 + j \cdot y1) \cdot (x2 + j \cdot y2) \\ &= (x1 \cdot x2 - y1 \cdot y2) + j \cdot (x1 \cdot y2 + x2 \cdot y1). \end{aligned}$$

Division of two complex numbers is a little more involved. First, we define the *complex conjugate*, z^*, of a number, $z = x + j \cdot y$, as one with the same real part, x, but whose imaginary part is $-y$. Thus, the complex conjugate of z is:

$$z^* = x - j \cdot y$$

Observe that the product of z and z^* is real:

$$\begin{aligned} z \cdot z^* &= (x \cdot x + y \cdot y) + j \cdot (x \cdot y - x \cdot y) \\ &= x \cdot x + y \cdot y. \end{aligned}$$

Now, we can divide two complex numbers:

$$\frac{z1}{z2} = \frac{x1 + j \cdot y1}{x2 + j \cdot y2}$$

To separate the result into real and imaginary parts, we first make the denominator real by multiplying both the numerator and the denominator by $z2^*$.

$$\begin{aligned} \frac{z1}{z2} &= \frac{z1 \cdot z2^*}{z2 \cdot z2^*} \\ &= \frac{(x1 + j \cdot y1) \cdot (x2 - j \cdot y2)}{x2 \cdot x2 + y2 \cdot y2} \\ &= \frac{x1 \cdot x2 + y1 \cdot y2}{x2 \cdot x2 + y2 \cdot y2} + j \cdot * \frac{-x1 \cdot y2 \cdot x2 \cdot y1}{x2 \cdot x2 + y2 \cdot y2} \end{aligned}$$

With this description of complex numbers and operations on them, we would like to develop programs that can work with them. Complex number is not a native data type in C, but we would like to represent complex numbers in a program as if it were. We will define an *abstract data type*, `complex`, using `typedef` and define functions to serve as operators on complex numbers.

We will represent complex numbers as ordered pairs of real and imaginary parts, (using rectangular form defined above), and implement the ordered pairs as structures. We will use `typedef` to define a data type, `rect`, for this structure. (We choose the name `rect` because `complex` data type with an identical structure is already defined in `math.h`. We can, of course, use the `complex` type defined in `math.h`, but we define a `rect` type to illustrate the use of `typedef`). Figure 15.9 shows this definition and the functions for addition and multiplication of complex numbers. We use type `double` in the structure `rect` for greater precision in computation. In a similar manner, it is easy to write the remaining functions for subtraction and division of two complex numbers. Implementation of these functions is left as an exercise.

```
/*   File: compdef.h */
struct rect {
     double real;
     double imag;
};

typedef struct rect rect;

/*   File: computil.h */
rect addc(rect z1, rect z2);
rect multc(rect z1, rect z2);

/*   File: computil.c */
#include <stdio.h>
#include <math.h>   /* math function protos: sqrt(), atan(), etc. */
#include "compdef.h"
#include "computil.h"

/*   Returns a sum of two complex numbers - rect form. */
rect addc(rect z1, rect z2)
{    rect z;

     z.real = z1.real + z2.real;
     z.imag = z1.imag + z2.imag;
     return z;
}

/* Returns a product of two complex numbers - rect form. */
rect multc(rect z1, rect z2)
{    rect z;

     z.real = z1.real * z2.real - z1.imag * z2.imag;
     z.imag = z1.real * z2.imag + z1.imag * z2.real;
     return z;
}
```

Figure 15.9: Complex number utility functions

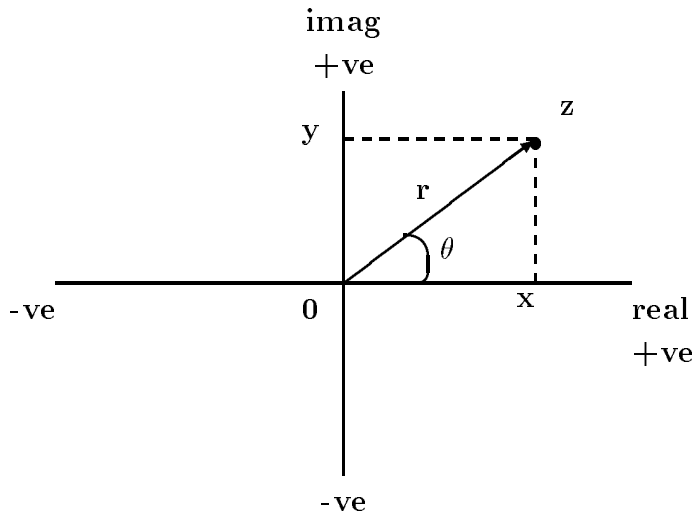

Figure 15.10: Complex Numbers in Polar Coordinates

15.2.1 Complex Numbers and Vectors

It is also possible to represent a point on a two dimensional plane in terms of polar coordinates. Polar coordinates are given in terms of:

1. the length, r, of the (directional) line from the origin to the point, and
2. the counterclockwise angle, θ, that the line makes with the reference axis, namely the positive horizontal axis.

The directional line of length r at an angle θ with respect to the reference axis is called a vector (See Figure 15.10). The projection of the vector onto the real axis is $r \cdot \cos(\theta)$, and the projection onto the imaginary axis is $r \cdot \sin(\theta)$. Thus, a complex number, represented by the pair (r, θ) in polar coordinates, can be written in rectangular coordinates as:

$$\begin{aligned} z &= r \cdot \cos(\theta) + j \cdot r \cdot \sin(\theta) \\ &= x + j \cdot y \end{aligned}$$

Thus, the real and imaginary parts, x and y, in terms of r and θ are:

$$\begin{aligned} x &= r \cdot \cos(\theta) \\ y &= r \cdot \sin(\theta) \end{aligned}$$

Since,

$$\exp(j \cdot \theta) = \cos(\theta) + j \cdot \sin(\theta)$$

z can also be written as:

$$z = r \cdot \exp(j \cdot \theta)$$

As we shall soon see, this exponential form is convenient for multiplication and division.

Given rectangular coordinates x and y, we can determine r and θ as follows. We know:

$$\begin{aligned} x^2 + y^2 &= r^2 \\ \frac{y}{x} &= \tan(\theta) \end{aligned}$$

so,

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \arctan(\frac{y}{x}) \end{aligned}$$

Observe that the length, r, is the square root of $z \cdot z^*$. The length r is called the *magnitude* of the vector, and the angle θ is called the *angle* or *phase angle* of the vector.

As we have seen, addition and subtraction of complex numbers is easy to perform in rectangular coordinates. On the other hand, multiplication and division of two complex numbers in rectangular coordinates is not so easy. Conversely, it is easy to perform multiplication and division in polar coordinates. Given that two numbers are:

$$\begin{aligned} p1 &= r1 * \exp(j \cdot \theta 1) \\ p2 &= r2 * \exp(j \cdot \theta 2) \end{aligned}$$

It is easy to see that:

$$\begin{aligned} p1 \cdot p2 &= r1 \cdot r2 \cdot \exp(j \cdot (\theta 1 + \theta 2)) \\ \frac{p1}{p2} &= \frac{r1}{r2} \cdot \exp(j \cdot (\theta 1 - \theta 2)). \end{aligned}$$

From this analysis, we can implement complex numbers in polar coordinates as shown in Figure 15.11 together with functions for multiplication and division in polar coordinates. It is also important to be able to convert back and forth between rectangular and polar coordinates. It is easy to write the necessary conversion routines to convert complex numbers in rectangular coordinates to polar coordinates, and vice versa — they are shown in Figure 15.12. The function `polar_to_rect()` is quite straight forward; `rect_to_polar()` uses the arc tangent function `atan()` defined in the standard library. This function returns an angle in the range $-\pi/2$ to $\pi/2$, thus we need to adjust the angle when the real part is zero and when it is negative. If the real part is zero, the angle is $\pi/2$ if the imaginary part is positive, and $-\pi/2$ if it is negative. Next, if the real part is negative, the angle must be incremented by π. Since we use many standard library trigonometric functions, the file `math.h` must be included at the head of `computil.c` and we must link the math library when the program is compiled.

These functions provide a useful library for processing with complex numbers. Let us now make use of them in two application programs.

15.2.2 Roots of Algebraic Equations

One such application where complex numbers occur is in finding roots of algebraic equations. A *linear* algebraic equation of the form:

$$a * x + b = 0$$

```
/* File:  compdef.h - continued */
struct polar {
     double r;
     double theta;
};

typedef struct polar polar;


/*   File: computil.h - continued */
polar multp(polar p1, polar p2);
polar divp(polar p1, polar p2);

/*   File:  computil.c - continued */
/*   Returns a product of complex numbers - polar form. */
polar multp(polar p1, polar p2)
{    polar p;

     p.r = p1.r * p2.r;
     p.theta = p1.theta + p2.theta;
     return p;
}

/* Returns p1 / p2 - polar form. */
polar divp(polar p1, polar p2)
{    polar p;

     p.r = p1.r / p2.r;
     p.theta = p1.theta - p2.theta;
     return p;
}
```

Figure 15.11: Complex number utility functions in polar coordinates

```
/*   File: computil.h - continued */
rect polar_to_rect(polar p);
polar rect_to_polar(rect z);

/* File: computil.c - continued */
/* Returns the rect form of a number in polar form. */
rect polar_to_rect(polar p)
{    rect z;

     z.real = p.r * cos(p.theta);
     z.imag = p.r * sin(p.theta);
     return z;
}

/* Returns the polar form of a number in rect form. */
#define PI 3.14159
polar rect_to_polar(rect z)
{    polar p;

     p.r = sqrt(z.real * z.real + z.imag * z.imag);
     if (z.real == 0)
          p.theta = z.imag >= 0 ? PI / 2 : - PI / 2;
     else
          p.theta = atan(z.imag / z.real);
     if (z.real < 0)
          p.theta = PI + p.theta;
     return p;
}
```

Figure 15.12: Conversion from polar to rect and rect to polar

in one unknown variable, x, can be easy to solve depending on the values of the coefficients, a and b. If $a = 0$ and $b = 0$, the equation is homogeneous and has no unique solution; any value for x will make the equation true. If $a = 0$ but b is non-zero, the equation has no solution; no value of x will make it true. Otherwise, if a is non-zero, the solution for x is easily determined:

$$x = -b/a$$

A *quadratic* equation is a polynomial of second degree in x of the form:

$$a \cdot x^2 + b \cdot x + c = 0$$

If a is zero, the equation reduces to a linear equation that is easy to solve. If a is non-zero, there are two solutions:

$$\begin{aligned} x1 &= \frac{-b + \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a} \\ x2 &= \frac{-b - \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a} \end{aligned}$$

The form of the solutions depends on the *discriminant*:

$$b^2 - 4 \cdot a \cdot c$$

If the discriminant is positive, the square root is a real number and the roots, $x1$ and $x2$ are both real numbers. If the discriminant is zero, the two roots are real and equal. Otherwise, if the discriminant is negative, the square root is an imaginary number and the roots are complex numbers:

$$\begin{aligned} x1 &= \frac{-b}{2 \cdot a} + j \cdot \frac{\sqrt{4 \cdot a \cdot c - b^2)}}{2 \cdot a} \\ x2 &= \frac{-b}{2 \cdot a} - j \cdot \frac{\sqrt{4 \cdot a \cdot c - b^2)}}{2 \cdot a} \end{aligned}$$

In fact, the two roots are complex conjugates: the real parts are the same, the imaginary parts are negatives of each other. Complex roots of polynomials with real coefficients always occur in complex conjugate pairs.

We will now implement a program that finds the roots of a quadratic equation, and then tests each root by evaluating the quadratic polynomial for that value of the variable. If the value is a root, the polynomial must evaluate to zero. When testing roots, we must be able to evaluate the polynomial for all possible values of roots, including complex values. For consistency in testing, we will represent all roots as complex numbers with real roots having a zero imaginary part. Therefore, we will need a function to force a real number into a complex number, as well as a function to make a complex number given its real and imaginary parts. These functions are shown in Figure 15.13, and are added to the file `computil.c` with their prototypes in `computils.h`. Finally, since complex numbers are not a native data type in C, we will also need a function to print complex numbers in the accepted form. If the number is real, it must print only the real part. If the number is imaginary, it must print only j times the imaginary part. Otherwise, it must print a complex number as $a + j \cdot b$ or $a - j \cdot b$, depending on the sign of the imaginary part. The function is also shown in Figure 15.13.

```
/*   File: computil.h - continued */
rect make_rect(double x, double y);
rect force_rect(double x);
void print_rect(rect z);

/*   File: computil.c - continued */
/*   Makes a complex number in rect form. */
rect make_rect(double x, double y)
{    rect z;

     z.real = x;
     z.imag = y;
     return z;
}

/*   Forces a real number to a complex number - rect form. */
rect force_rect(double x)
{    rect z;

     z.real = x;
     z.imag = 0;
     return z;
}

/*   Prints a complex number in rect form. */
void print_rect(rect z)
{
     if (z.real == 0 && z.imag == 0)    /* if number is zero */
          printf("0");                  /* print zero. */
     if (z.real != 0)              /* print real part, if non-zero */
          printf("%f ", z.real);
     if (z.imag != 0) {            /* print imag part, if non-zero */
          if (z.imag > 0)
               printf("+ j * %f", z.imag);
          else if (z.imag < 0)
               printf("- j * %f", -z.imag);
     }
     printf("\n");
}
```

Figure 15.13: Code for `make_rect()` and `force_rect()`

With all of these utility functions completed, the program logic is now simple to implement. It reads in the coefficients a, b, c of the quadratic equation and uses the function `findroots()` to find the roots of the quadratic. The function forces the roots to complex form and returns them indirectly. The arguments of `findroots()` are the coefficients of the quadratic, and pointers to the two roots. The program then uses the function `eval_quad()` to verify each root by evaluating the quadratic polynomial at that value. The arguments of `eval_quad()` are the coefficients of the quadratic, and the value at which the quadratic is to be evaluated. The code for the driver is shown in Figure 15.14. For each set of coefficients, `main()` checks if a is zero and b is non-zero, in which case it prints that the equation is linear with root `-c/b`. Otherwise, if both a and b are zero, it prints an invalid equation message, and in either case continues to read the next set of coefficients. On the other hand, if a is non-zero, the driver calls `findroots()` to find the roots as complex numbers and returns them by indirectly to `z1` and `z2`. Each root is printed and verified using `eval_quad()`. The process continues until end of file.

We next implement the function `findroots()` shown in Figure 15.15. It computes the roots, forces them to complex numbers and returns the values through the pointer parameters.

Finally, we write `eval_quad()` to evaluate a quadratic polynomial at a given complex value of the unknown variable. Since the value of the unknown, x is complex, we force all coefficients to complex numbers before using our utility functions `addc()` and `multc()`. To reduce the number of multiplications required to evaluate the polynomial we perform the expression

$$\begin{aligned} a \cdot x^2 + b \cdot x + c &= a \cdot x \cdot c + b \cdot x + c \\ &= (((a \cdot z) + b) \cdot z) + c \end{aligned}$$

The function is shown in Figure 15.16 The complex variable, `w`, is initialized to zero and then used for the cumulative complex sum of the polynomial. As we saw in Chapter 5, due to errors in rounding and floating point number representation, our result may not be precisely zero. Therefore, `eval_quad()` checks that `w.real` and `w.imag` are sufficiently close to zero using the library function `fabs()` to verify the solution and print an appropriate message. A sample run of the program is shown below:

```
***Roots of Quadratic Equations***

Quadratic Equation:  a * x * x + b * x + c = 0
Type coefficients a b c, EOF to quit
2 3 5
z1 = -0.750000 + j * 1.391941
The value is verified as a root of the equation
z2 = -0.750000 - j * 1.391941
The value is verified as a root of the equation
1 2 1
z1 = -1.000000
The value is verified as a root of the equation
z2 = -1.000000
The value is verified as a root of the equation
2 2 5
z1 = -0.500000 + j * 1.500000
The value is verified as a root of the equation
```

```
/*   File; roots.c
     Other Source Files: computil.c
     Header Files: compdef.h, computil.h
     This program finds the roots of quadratic equations. For each
     equation, the program verifies that the roots make the
     quadratic polynomial evaluate to zero. All roots, including real
     roots, are treated as complex roots.
*/

#include <stdio.h>
#include <math.h>        /* needed in this file and in computil.c */
#include "compdef.h"     /* defines rect and polar types */
#include "computil.h"    /* prototypes for functions in computil.c */

void eval_quad(double a, double b, double c, rect z);
void findroots(double a, double b, double c, rect *zp1, rect *zp2);

main()
{    rect z1, z2;
     double a, b, c, x;

     printf("***Roots of Quadratic Equations***\n\n");
     printf("Quadratic Equation: a * x * x + b * x + c = 0\n");
     printf("Type coefficients a b c, EOF to quit\n");
     while (scanf("%lf %lf %lf", &a, &b, &c) != EOF) {
          if (a == 0) {
               if (b != 0) {
                    printf("Linear equation - root is %f\n", - c / b);
                    continue;
               }
               else {
                    printf("Invalid equation\n");
                    continue;
               }
          }
          else
               findroots(a, b, c, &z1, &z2);
          printf("z1 = ");
          print_rect(z1);
          eval_quad(a, b, c, z1);
          printf("z2 = ");
          print_rect(z2);
          eval_quad(a, b, c, z2);
     }
}
```

Figure 15.14: Code for quadratic solver driver

```
/*   File; roots.c - continued */
/*   Finds the roots of a quadratic equation. Roots are forced
     to complex values and stored where zp1 and zp2 point.
*/
void findroots(double a, double b, double c, rect *zp1, rect *zp2)
{    double discr, x, x1r, x2r, x1i, x2i;
     rect z1, z2;

     x = 2 * a;
     discr = b * b - 4 * a * c;
     if (discr >= 0) {
          x1r = -b / x + sqrt(discr) / x;
          x2r = -b / x - sqrt(discr) / x;
          x1i = x2i = 0;
     }
     else {
          x1r = x2r = -b / x;
          x1i = sqrt(-discr) / x;
          x2i = -x1i;
     }
     z1 = make_rect(x1r, x1i);
     z2 = make_rect(x2r, x2i);
     *zp1 = z1;
     *zp2 = z2;
}
```

Figure 15.15: Code for `findroots()`

```
/*   File; roots.c - continued */
/*   Function evaluates a quadratic equation with x equal to
     the unknown variable.
*/
void eval_quad(double a, double b, double c, rect x)
{    rect w = {0, 0};

     w = multc(force_rect(a), x);  /* a * x */
     w = addc(w, force_rect(b));   /* a * x + b */
     w = multc(w, x);              /* a * x * x + b * x */
     w = addc(w, force_rect(c));   /* a * x * x + b * x + c */
     if (fabs(w.real) < 0.000001 && fabs(w.imag) < 0.000001)
          printf("The value is verified as a root of the equation\n");
     else
          printf("The value is not a root of the equation\n");
}
```

Figure 15.16: Code for `eval_quad()`

```
z2 = -0.500000 - j * 1.500000
The value is verified as a root of the equation
^D
```

15.2.3 Impedance of Electrical Circuits

Another important application of complex numbers is in computing impedances of electrical circuits. The basic components of such circuits are resistors, inductors, and capacitors as shown in Figure 15.17. These devices can be connected in series or parallel to make more complex circuits as shown in Figure 15.18 where each component has an impedance, Z. In general, the impedance is modeled as a complex quantity depending on their value and the value of the angular frequency, ω, in radians per second, of the electrical signal for which the impedance is to be computed. The impedance of a resistor of R ohms is simply R, that of an inductor of L henrys is $j \cdot \omega \cdot L$, and that of a capacitor of C farads is $\frac{-j}{(\omega \cdot C)}$.

The impedance of a series or a parallel combination of sub-circuits is defined in terms of the individual impedances of the sub-circuits. The impedance of a series combination of impedances, *Z1* and *Z2* is the sum of the individual impedances, i.e. $Z1 + Z2$. The impedance of a parallel combination of impedances *Z1* and *Z2* is the reciprocal of the reciprocal sum of the individual impedances:

$$\frac{1}{\frac{1}{Z1} + \frac{1}{Z2}}$$

Let us first write a set of circuit utility functions to determine:

- the impedance of a basic component,
- the impedance of a series combination,

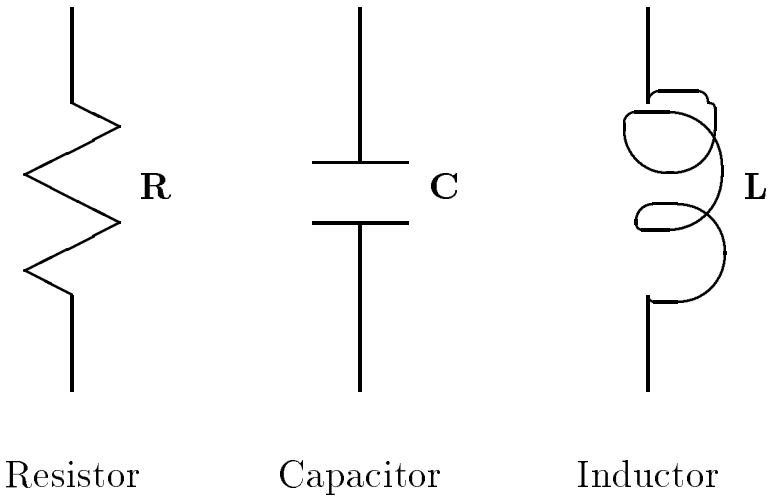

Figure 15.17: Basic Electrical Circuit Components

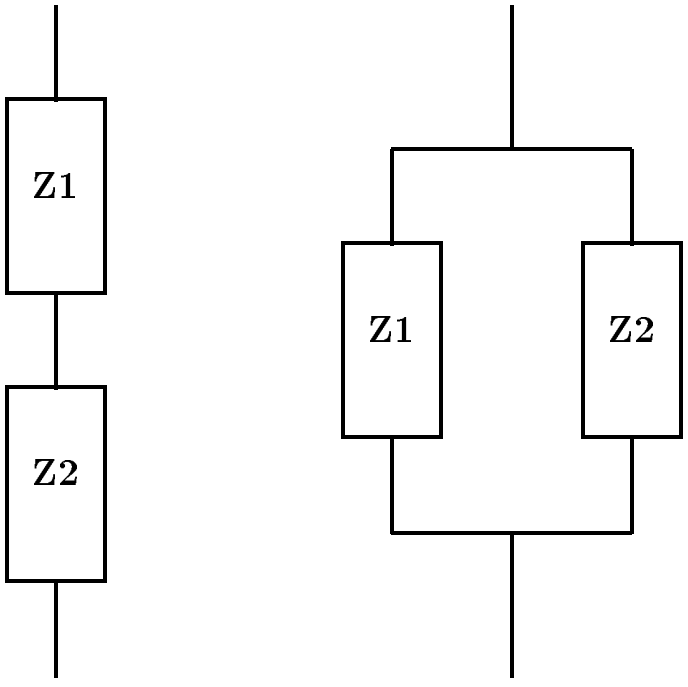

Figure 15.18: Series and Parallel Combinations

- and the impedance of a parallel combination.

We will use the complex data type, `rect`, as defined in `compdef.h` as well as the functions defined in `computil.c`.

The function `comp_imped()` determines the impedance of a basic component whose element type (a character) and value are passed together with the value of the angular frequency ω (we will call `w`). The code is shown in Figure 15.19. The only point to note here is that if `w * C` is zero, the impedance is infinite. It is not possible to handle an infinite value in computers, so some garbage value is returned. The calling program must handle a zero value of `w * C` as a special case. Next, we implement the functions that compute the series and parallel combination of impedances shown in Figure 15.20. The function `series()` merely returns the sum of the two impedances. The function `parallel()` uses polar coordinates to compute the reciprocals of impedances. It is much easier to compute the reciprocal of a complex number in polar coordinates than in rectangular coordinates; whereas complex numbers are easier to sum in rectangular coordinates. Conversion routines are used to convert polar to rectangular, and vice versa.

*

We are now ready to implement a program to compute the impedance of an electrical circuit. Let us assume a circuit which is a series combination of two sub-circuits as shown in Figure 15.21. The first sub-circuit is a series combination of resistor **R1** and inductor **L**. The second sub-circuit is a parallel combination of resistor **R2** and capacitor **C**. Figure 15.22 shows the program to find the impedances of this circuit for different sets of values of **R1**, **R2**, **L**, **C**, and ω. The program reads a set of values for **R1**, **R2**, **L**, **C**, and ω. It calls `series()` to compute the impedance `z1` of **R1** and **L** in series. If ωC is zero, the impedance of the capacitor is infinite; so the impedance of the parallel combination, `z2`, is just the impedance of **R2**. Otherwise, `parallel()` is called to compute the impedance `z2` of **R2** and **C** in parallel. In all cases, `comp_imped()` is used to compute the impedances of the basic components and `series()` is called to compute the impedance of `z1` and `z2` in series. The values of these impedances are printed by `print_rect()`. A sample run is shown below:

```
***Impedance of Electrical Circuits***

Ckt:  a series combination of:
R1 and L in series, and
R2 and C in parallel.
Type values of R1 R2 L C W, EOF to quit
1 1 1 1 1
Impedance of series branch z1 = 1.000000 + j * 1.000000
Impedance of parallel branch z2 = 0.500000 - j * 0.499999
Overall impedance z = 1.500000 + j * 0.500001
10 10000 0.01 0.000001 10000
Impedance of series branch z1 = 10.000000 + j * 100.000000
Impedance of parallel branch z2 = 1.000023 - j * 99.990005
Overall impedance z = 11.000023 + j * 0.009995
^D
```

The second circuit values represent a circuit near resonance. Its impedance is almost purely resistive, since the imaginary part is close to zero.

```
/*   File: cktutil.h */
rect comp_imped(int component, double value, double w);

/*   File: cktutil.c */
#include <stdio.h>
#include <math.h>
#include "compdef.h"
#include "computil.h"
#include "cktutil.h"
/*   Returns the impedance for each of the components R, L, C. */
rect comp_imped(char component, double value, double w)
{    rect z;
     double x;

     switch(component) {
          case 'r': z = force_rect(value);   /* impedance is R */
                    break;
          case 'l': z.real = 0;
                    z.imag = w * value;      /* impedance is j * w * L */
                    break;
          case 'c': z.real = 0;
                    x = w * value;           /* x = w * C */
                    /* if x is non-zero, impedance is -j/(w*C) */
                    if (x)
                         z.imag = - 1 / x;
                    else ;              /* else, impedance is infinite */
                    break;              /* handle separately */
     }
     return z;
}
```

Figure 15.19: Code for `comp_imped()`

```
/*   File: cktutil.h - continued */
rect series(rect z1, rect z2);
rect parallel(rect z1, rect z2);

/*   File: cktutil.c - continued */
/*   Returns the impedance of a series combination of impedances
     z1 and z2: sum of z1 and z2.
*/
rect series(rect z1, rect z2)
{
     return addc(z1, z2);
}

/*   Returns the impedance of a parallel combination of impedances
     z1 and z2: reciprocal of the sum of 1 / z1 and 1 / z2.
*/
rect parallel(rect z1, rect z2)
{    polar p1, p2, p;
     rect z;

     p1 = rect_to_polar(z1);
     p1.r = 1 / p1.r;              /* reciprocal of z1 */
     p1.theta = -p1.theta;
     p2 = rect_to_polar(z2);
     p2.r = 1 / p2.r;              /* reciprocal of z2 */
     p2.theta = -p2.theta;
     z = addc(polar_to_rect(p1), polar_to_rect(p2));   /* sum reciprocals
*/
     p = rect_to_polar(z);
     p.r = 1 / p.r;                /* take reciprocal of the sum */
     p.theta = -p.theta;
     z = polar_to_rect(p);         /* convert to rect. */
     return z;                     /* return in rect form */
}
```

Figure 15.20: Code for `series()` and `parallel()`

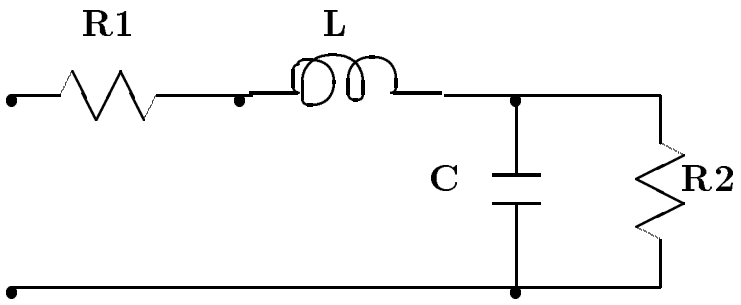

Figure 15.21: An Example Circuit

15.3 Integrals

Another common operation that arises in engineering and scientific computing is integration. While software application packages exist to perform symbolic integration, some functions do not lend themselves to such a "closed form" method. A common computing method for approximating the value of an integral in *numeric integration*. In this section we will develop a small program implementing Simpson's Rule for numeric integration.

The integral of a function between specified limits gives the area under the function curve as shown in Figure 15.23. Numeric methods can approximate the area under the curve by summing approximate sub-areas under linearized parts of the function at uniformly sampled points (Figure 15.24). The smaller the sampling interval, h, the greater the precision of the computed integral. An algorithm to evaluate such an integral may be written in terms of the value of the function at sample points between the two limits. For example, assume the limits of integration for function, $f(x)$ are $x = a$ and $x = b$. Then, the function values between the two limits at intervals of h are:

$$f(a), f(a+h), f(a+2h), \ldots, f(b)$$

The total number of samples is

$$\frac{(b-a)}{h}$$

There are many methods to approximate the value of an integral in terms of these sample values. Simpson's Rule gives a fairly accurate integral of function $f(x)$ between specified limits a and b:

$$Integralvalue = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + y_n)$$

where, $y_k = f(a+kh)$ for $k = 0, 1, 2, ..., n$, h is the sampling interval, and $n = \frac{b-a}{h}$ with h adjusted so that n is an even integer. Except for the multiplier $\frac{h}{3}$, the above sum is called the *Simpson sum*. Observe that in the Simpson sum, sample values of the function evaluated at odd k, i.e. $y_1, y_3, y_5, \ldots$, are multiplied by 4, and sample values at even values of k, except for y_0 and y_n, are multiplied by 2. Finally, sample values y_0 and y_n are added without a multiplier.

We will now slightly modify the concept of a generic sum from Chapter 14 to implement a function that numerically evaluates an integral of a specified function between two limits, i.e. modify the generic function, `sum()`, into a generic Simpson sum function. Since integral computation requires real numbers, we use type `double` for all our computation. The parameters to `simpsum`

```
/*   File; imped.c
     Other Source Files: computil.c, cktutil.c
     Header Files: compdef.h, computil.h, cktutil.h
     This program finds the impedance of an electrical circuit for
     different values of the components and the frequency. The
     circuit consists of a series of two sub-circuits: a series
     combination of a resistor R1 and an inductor L, and a parallel
     combination of a resistor R2 and a capacitor C. The values of
     these components are specified by the user together with the
     angular frequency w in radians per second. The impedance is found
     for each user specified set of values until EOF.
*/
#include <stdio.h>
#include <math.h>
#include "compdef.h"
#include "computil.h"
#include "cktutil.h"
main()
{    rect z, z1, z2;
     double r1, r2, l, c, w;

     printf("***Impedance of Electrical Circuits***\n\n");
     printf("Ckt: A series combination of:\n");
     printf("        R1 and L in series, and\n");
     printf("        R2 and C in parallel.\n");
     printf("Type values of R1 R2 L C w, EOF to quit\n");

     while (scanf("%lf %lf %lf %lf %lf",
                  &r1, &r2, &l, &c, &w) != EOF) {
          z1 = series(comp_imped('r', r1, w), comp_imped('l', l, w));
          if (w == 0 || c == 0)
               z2 = comp_imped('r', r2, w);
          else
               z2 = parallel(comp_imped('r', r2, w),
                             comp_imped('c', c, w));
          z = series(z1, z2);
          printf("Impedance of series branch z1 = ");
          print_rect(z1);
          printf("Impedance of parallel branch z2 = ");
          print_rect(z2);
          printf("Overall impedance z = ");
          print_rect(z);
     }
}
```

Figure 15.22: Driver program for an example circuit

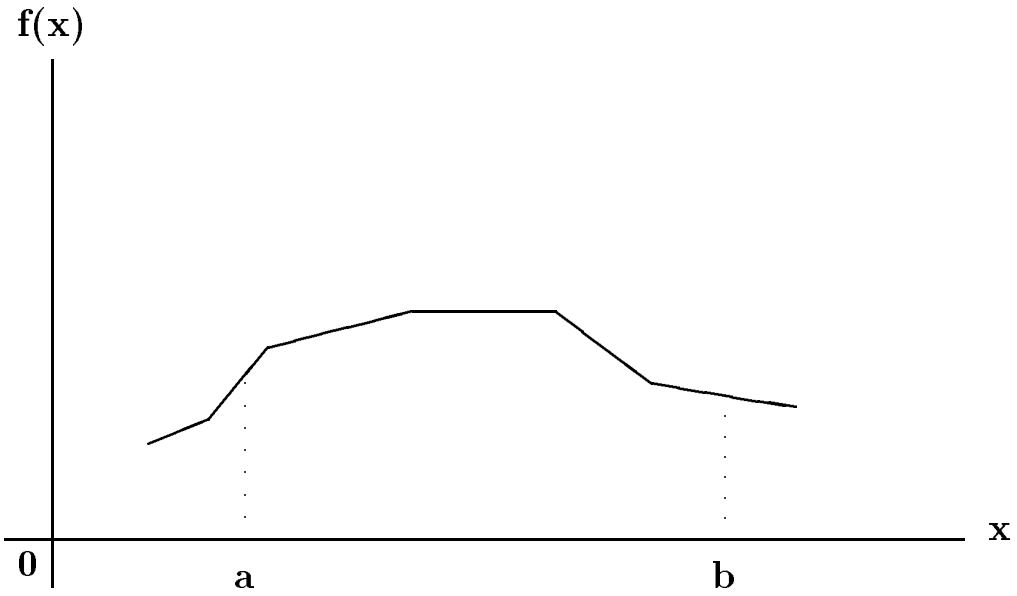

Figure 15.23: Integral of a Function from a to b

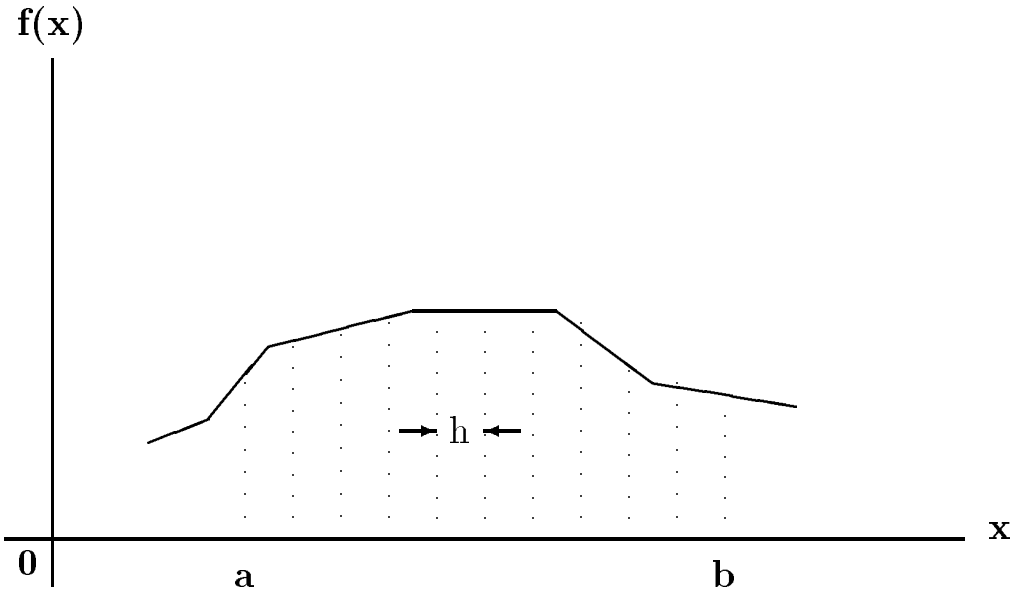

Figure 15.24: Function Sampling at Intervals of h

```
/*   File: simputil.h */
double simpsum(double (*fp)(), double a, double (*up)(),
               double step, double b);

/*   File: simputil.c */
#include <stdio.h>
#include "simputil.h"
/*   Returns the Simpson sum of *fp from a to b. */
double simpsum(double (*fp)(), double a, double (*up)(),
                    double step, double b)
{    double i, cumsum = 0;
     int m;

     for (i = a, m = 0; i < b; m++, i = (*up)(i, step)) {
          if (m == 0)
               cumsum += (*fp)(i);
          else if (m % 2)
               cumsum += 4 * (*fp)(i);
          else
               cumsum += 2 * (*fp)(i);
     }
     cumsum += (*fp)(b);
     return cumsum;
}
```

Figure 15.25: Code to compute the Simpson sum

are the function pointer, `fp`, a lower limit, `a`, an update function pointer, `up`, a sampling interval, `step`, and an upper limit, `b`. The code is shown in Figure 15.25.

The integer variable, `m` represents the sample number. If `m` is zero, the function sample is added to the cumulative sum, if it is odd, the sample value times 4 is added to the cumulative sum; otherwise, sample value times 2 is added. Finally, the sample value y_n at `b` is added and the resulting Simpson sum is returned. It is easy now to implement the function `integral()` to compute the integral of a function between limits `a` and `b`. It merely gets the Simpson sum, multiplies by `step/3` and returns it as seen in Figure 15.26 The update function `incr()` merely returns the value of its first argument increased by the value of the second argument, `step`. This function is included in `sumutil.c` together with other useful functions, `self()`, `square()`, and `cube()` shown in Figure 15.27.

Finally, we write a simple driver that computes integrals of several functions using `integral()` shown in Figure 15.28. The program first reads the sampling interval, `h`; then repeatedly reads the integration limits until `EOF`. For each set of limits, it calculates the number of samples, `n`, for the specified `h`. Since the Simpson sum requires an even number of samples, `n` is increased by one if it is odd, and the sampling interval `h` is adjusted to correspond to the even value of `n`. Then, the program computes the integral by calling `integral()` for three different functions: a straight

```
/*   File: simputil.h - continued */
double integral(double (*fp)(), double a, double b, double step);

/*   File: simputil.c - continued */
/*   Computes integral of a function *fp from a to b in sample
     steps of step.
*/
double integral(double (*fp)(), double a, double b, double step)
{    double r, incr();

     r = simpsum(fp, a, incr, step, b);
     return r * step / 3;
}
```

Figure 15.26: Code for `integral()`

line $f(x) = x$, a square, $f(x) = x^2$, and a cube, $f(x) = x^3$. The values of integrals are printed. The program is in three source files, which must be compiled and linked: `integr.c`, `sumutil.c`, and `simputil.c`. Here are two sample sessions with different sampling intervals:

```
***Integration by Simpson's Rule***

Integrals of x, square of x, and cube of x
Sampling interval for integration:  0.1
Type lower and upper limits, EOF to quit
0 1
Integral of st.  line = 0.566667
Integral of square = 0.400000
Integral of cubic = 0.316667
^D

***Integration by Simpson's Rule***

Integrals of x, square of x, and cube of x
Sampling interval for integration:  0.01
Type lower and upper limits, EOF to quit
0 1
Integral of st.  line = 0.500000
Integral of square = 0.333333
Integral of cubic = 0.250000
^D
```

Remember, the smaller the sampling interval, the greater the accuracy of the computed integral. The first session specifies a fairly large sampling interval of 0.1 and the results are not very accurate. The exact answers for the integrals are 0.5, 0.3333, and 0.25. The second session specifies a

```
/*   File: sumutil.h - continued */
double self(double x);
double square(double x);
double cube(double x);
double incr(double x, double step);

/*   File: sumutil.c - continued */
/*   Returns x. */
double self(double x)
{
     return x;
}

/*   Returns square of x. */
double square(double x)
{
     return x * x;
}

/*   Returns cube of x. */
double cube(double x)
{
     return x * x * x;
}

/*   Returns x incremented by step. */
double incr(double x, double step)
{
     return x + step;
}
```

Figure 15.27: Code for `self()`, `cube()`, and `incr()`

```
/*   File: integr.c
     Other Source Files: sumutil.c, simputil.c
     Header Files: sumutil.h, simputil.h
     This program computes definite integrals of several functions
     between specified limits. Parameters of integral() are: a
     function pointer, limits, and number of samples. It returns
     the integral of that function. Integrals of straight line,
     square, and a cubic are printed out for specified limits.
     Simpson's Rule is used to compute integral of a function f(x)
     between limits a and b as follows:

          I = (h / 3) * (y0 + 4y1 + 2y2 + 4y3 + 2y4 +...+ yn),

     where, yk = f(a + kh), and h = (b - a) / n for some even integer
     n. Except for the multiplier h/3, the above sum is called the
     Simpson sum.
*/

#include <stdio.h>
#include "sumutil.h"
#include "simputil.h"

main()
{    double  r, a, b, h, self(), square(), cube();
     int n;

     printf("***Integration by Simpson's Rule***\n\n");
     printf("Integrals of x, square of x, and cube of x\n");
     printf("Sampling interval for integration: ");
     scanf("%lf", &h);
     printf("Type lower and upper limits, EOF to quit\n");
     while (scanf("%lf %lf", &a, &b) != EOF) {
          n = (b - a) / h;
          if (n % 2) {
               n++;
               h = (b - a) / n;
          }
          r = integral(self, a, b, h);
          printf("Integral of st. line = %f\n", r);
          r = integral(square, a, b, h);
          printf("Integral of square = %f\n", r);
          r = integral(cube, a, b, h);
          printf("Integral of cubic = %f\n", r);
     }
}
```

Figure 15.28: Driver for Numeric Integration Program

somewhat better sampling interval 0.01, and the results are quite accurate. A smaller sampling interval would be even better, but would require more computation time. A compromise between accuracy and speed is required in most numeric computations.

15.4 Summary

In this chapter we have shown how we can use the features of the C language as well as the program design techniques we have discussed throughout this text to implement programs for several common engineering and scientific applications. In general, we have done this by developing a set of utility functions to use as a toolbag for writing the application. Our treatment of engineering and scientific computing has not been, by any means, comprehensive. References, such as [1] below, can be a source of algorithms for many additional applications. However, with your current knowledge of C you can now develop programs from these algorithms to solve **your** problems.

E hoʻomaikaʻi ʻoukou.
(Good luck).

References:

[1]. Press, William H., Flannery, Brian P., Teukolsky, Saul A., Vettering, William T., Numerical Recipes in C, Cambridge University Press, Cambridge, 1988.

15.5 Problems

1. Write a menu driven program that allows the user to specify a matrix operation: add, subtract, multiply.

2. Write a simple calculator program that performs complex number arithmetic. The input should be an operand, followed by an operator, followed by an operand. The output should be the result of applying the operator to operands.

3. Repeat 2, but allow the user to continue entering operators and operands in sequence. The user may also request that a value should be saved for later use.

4. Evaluate a polynomial $P(z)$ with specified coefficients for a complex value of the variable. The highest degree of the polynomial is 10. The user must enter coefficient and exponent pairs for the polynomial, and specify the value of the variable for which the polynomial is to be evaluated.

5. Consider a rational function of a variable s:

$$\frac{P(s)}{Q(s)}$$

   where $P(s)$ and $Q(s)$ are polynomials in a variable, s, with real coefficients. Evaluate the function for a value of $s = j\omega$. Evaluate the function for different values of s. Plot the magnitude and angle of the values.