This chapter describes why IEEE standard floating-point arithmetic is important and why you should use it when programming. PowerPC Numerics is an implementation of the IEEE Standard 754 for binary floating-point arithmetic as well as the standard proposed by the Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group (NCEG). This chapter explains the benefits that PowerPC Numerics provides by conforming to these standards. It provides an overview of both the IEEE and the FPCE recommendations—describing the scope of these standards and explaining how following them improves the accuracy of your programs. It provides some examples to demonstrate how much easier programming is when the standards are followed. Finally, it describes in general how PowerPC Numerics differs from the Standard Apple Numerics Environment (SANE).

You should read this chapter if you are unfamiliar with IEEE Standard 754 or the FPCE technical report and you want to find out more about them. If you are already familiar with these standards but you would like to find out how PowerPC Numerics implements them, you can skip to the next chapter.

About the IEEE Standard

PowerPC Numerics is a floating-point environment that complies with IEEE Standard 754. There are two IEEE standards for floating-point arithmetic: **IEEE Standard 754** for binary floating-point arithmetic and **IEEE Standard 854** for radix-independent floating-point arithmetic. When you see the term **IEEE standard** in this book without a number following, it means IEEE Standard 754.

The IEEE standards ensure that computers represent real numbers as accurately as possible and that computers perform arithmetic on real numbers as accurately as possible. Although there are infinitely many real numbers, a computer can represent only a finite number of them. Computers represent real numbers as **binary floating-point numbers**. Binary floating-point numbers can represent real numbers exactly in relatively few cases; in all other cases the representation is approximate. For example, 1/2 (0.5 in decimal) can be represented exactly in binary as 0.1. Other real numbers that can be represented exactly in decimal have repeating digits in binary and hence cannot be represented exactly, as shown in Table 1-1. For example, 1/10, or decimal 0.1 exactly, is 0.000110011 . . . in binary. Errors of this kind are unavoidable in any computer approximation of real numbers. Because of these errors, sums of fractions are often slightly incorrect. For example, 4/3 – 5/6 is not exactly equal to 1/2 on any computer, even on computers that use IEEE standard arithmetic.

Table 1-1 Approximation of real numbers

Fraction	Decimal approximation*	Binary approximation†
1/10	0.1000000000‡	0.0001100110011001100110011001101
1/2	0.5000000000‡	0.10000000000000000000000000‡
4/3	1.333333333	1.01010101010101010101011
5/6	0.8333333333	0.11010101010101010101010101
4/3 – 5/6	0.4999999997	0.10000000000000000000000001

* 10 significant digits
† 23 significant bits
‡ Exact value

The IEEE standard defines data formats for floating-point numbers, shows how to interpret these formats, and specifies how to perform operations (known as **floating-point operations**) on numbers in these formats. It requires the following types of floating-point operations:

- basic arithmetic operations (add, subtract, multiply, divide, square root, remainder, and round-to-integer)
- conversion operations, which convert numbers to and from the floating-point data formats
- comparison operations, such as less than, greater than, and equal to
- environmental control operations, which manipulate the floating-point environment

The IEEE standard requires that the basic arithmetic operations have the following attributes:

- The result must be accurate in the precision in which the operation is performed. When a numerics environment is performing a floating-point operation, it calculates the result to a predetermined number of binary digits. This number of digits is called the **precision.** The result must be correct to the last binary digit.
- If the result cannot be represented exactly in the destination data format, it must be changed to the closest value that can be represented, using **rounding.** See the section “Careful Rounding” on page 1-5 for more information on why careful rounding is important.
- If an invalid input is provided or if the result cannot be represented exactly, a floating-point **exception** must be raised. See the section “Exception Handling” on page 1-6 for a description of why exception handling is important in floating-point arithmetic.

Starting to Use IEEE Arithmetic

You can get the benefit of much of the IEEE standard without special programming techniques; you simply use the floating-point variable formats and operations available in the programming language in which you are working, and the computer takes care of the rest. Other features might require changes to your applications. If you are new to numerical programming, you should approach the IEEE standard features in three stages:

1. Recompile your old programs with no changes; you will get many of the benefits.
2. Make small changes to obtain more benefits. For example, at this stage you might remove all code that tests for division by zero.
3. Use the advanced features, such as environmental controls, for special applications.

If you already use the IEEE standard features but your application is written for a non-Macintosh computer, see Appendix B, “Porting Programs to PowerPC Numerics.”

Careful Rounding

If the result of an IEEE arithmetic operation cannot be represented exactly in binary format, the number is rounded. IEEE arithmetic normally rounds results to the nearest value that can be represented in the chosen data format. The difference between the exact result and the represented result is the **roundoff error.**

The IEEE standard requires that users be able to choose to round in directions other than to the nearest value. For example, sometimes you might want to know that rounding has not invalidated a computation. One way to do that would be to force the rounding direction so that you can be sure your results are higher (or lower) than the exact answer. Because it conforms to the IEEE standard, PowerPC Numerics gives you a means of doing that. Fully developed, this strategy is called *interval arithmetic* (Kahan 1980). For complete details on rounding directions, see Chapter 4, “Environmental Controls.”

The following example is a simple demonstration of the advantages of careful rounding. Suppose your application performs operations that are mutually inverse; that is, operations $y = f(x)$, $x = g(y)$, such that $g(f(x)) = x$. There are many such operations, such as

$$y = x^2, \quad x = \sqrt{y}$$

$$y = 375x, \quad x = y/375$$

Suppose $F(x)$ is the computed value of $f(x)$, and $G(y)$ is the computed value of $g(y)$. Because many numbers cannot be represented exactly in binary, the computed values $F(x)$ and $G(y)$ will often differ from $f(x)$ and $g(y)$. Even so, if both functions are continuous and well behaved, and if you always round $F(x)$ and $G(y)$ to the nearest value, you might expect your computer arithmetic to return x when it performs the cycle of inverse operations, $G(F(x))$. It is difficult to predict when this relation will hold for computer numbers. Experience with other computers says it is too much to expect, but IEEE arithmetic very often returns the correct inverse value.

The reason for IEEE arithmetic's good behavior with respect to inverse operations is that it rounds so carefully. Even with all operations in, say, single precision, it evaluates the expression $3 \times 1/3$ to 1.0 exactly; some computers that do not follow the standard do not evaluate this expression exactly. If you find that surprising, you might enjoy running the code example in Listing 1-1 on a computer that does not use IEEE arithmetic and then on a PowerPC processor-based Macintosh computer. The default rounding provided by the numerics environment gives good results; the PowerPC processor-based Macintosh computer prints "No failures." The program will fail on a computer that doesn't have IEEE arithmetic—in particular, that doesn't round halfway cases in the same way that the IEEE standard's default rounding direction mode does.

Listing 1-1 Inverse operations

```
#include <stdio.h>
main()
{
   float x, y, a, b;
   int ix, iy,
   int nofail = 1;      /* Boolean, initialized to true */

   for (ix = 1; ix <= 12; ix++) {
      if ((ix != 7) && (ix != 11)) {      /* x is a sum of powers of two */
         for (iy = 1; iy <= 50; iy++) {
            x = ix;
            y = iy;
            a = y / x;
            b = x * a;        /* b == (x * y / x) == y */
            if (b != y) {
               nofail = 0;    /* false */
               printf("It failed for x = %d, y = %d\n", ix, iy);
            }
         }
      }
   }
   if (nofail) printf("No failures\n");
}
```

Exception Handling

The IEEE standard defines five exceptions that indicate when an exceptional event has occurred. They are

- invalid operation
- underflow

- overflow
- division by zero
- inexact result

There are three ways your application can deal with exceptions:

- Continue operation.
- Stop on exceptions if you think they will invalidate your results.
- Include code to do something special when exceptions happen.

The IEEE standard lets programs deal with the exceptions in reasonable ways. It defines the special values NaN (Not-a-Number) and Infinity, which allow a program to continue operation; see the section "Interpreting Floating-Point Values" in Chapter 2, "Floating-Point Data Formats." The IEEE standard also defines exception flags, which a program can test to detect exceptional events.

IEEE arithmetic allows the option to stop computation when exceptional events arise, but there are good reasons why you might prefer not to have to stop. The following examples illustrate some of those reasons.

Example: Finding Zero Return Values

Suppose you want to find the first positive integer that causes a function to cross the x-axis. A simple version of the code might look like this:

```
for (i = 0; i < MAXVALUE; i++)
   if (func(i) == 0)
      printf("It crosses when x = %g\n", i);
```

Further, suppose that `func` was defined like this:

```
double func(double x)
{
   return(sqrt(x - 3));
}
```

The intent of the `for` loop is to find out where the function crosses the x-axis and print out that information; it does not really care about the value returned from `func` unless the value is 0. However, this loop will fail when `i` is less than 3 because you cannot take the square root of a negative number. With a C compiler that supports PowerPC Numerics, performing the square root operation on a negative number returns a NaN, allowing the loop to produce the desired result. To obtain the desired result on all computers, something more cumbersome would have to be written. By allowing the square root of a negative number, PowerPC Numerics allows more straightforward code.

This program fragment demonstrates the principal service performed by NaNs: they permit deferred judgments about variables whose values might be unavailable (that is, uninitialized) or the result of invalid operations. Instead of having the computer stop a computation as soon as a NaN appears, you might prefer to have it continue if whatever caused the NaN is irrelevant to the solution.

Example: Searching Without Stopping

Suppose a program has to search through a database for a maximum value that has to be calculated. The search loop might call a subroutine to perform some calculation on the data in each record and return a value for the program to test or compare. The code might look like this:

```
max = –INFINITY;
for (i = 0; i < MAXRECORDS; i++)
   if((temp = computation(record[i].value)) > max)
      max = temp;
```

Suppose that the `value` field of the `record` structure is not a required field when the data is entered, so that for some records, data might be nonexistent or invalid. In many machines, that would cause the program to stop. To avoid having the program stop during the search, you would have to add tests for all the exceptional cases. With PowerPC Numerics, the subroutine `computation` does not stop for nonexistent or invalid data; it simply returns a NaN.

This is another example of the way arithmetic that includes NaNs allows the program to ignore irrelevancies, even when they cause invalid operations. Using arithmetic without NaNs, you would have to anticipate all exceptional cases and add code to the program to handle every one of them in advance. With NaNs, you can handle all exceptional cases after they have occurred, or you can simply ignore them, as in this example.

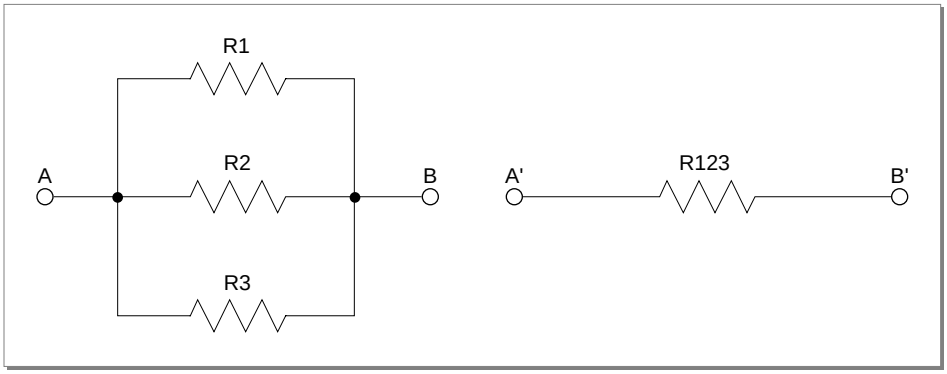
Example: Parallel Resistances

Like NaNs, Infinities enable the program to handle cases that otherwise would require special programming to keep from stopping. Here is an example where arithmetic with Infinities is entirely reasonable.

When three electrical resistances R1, R2, and R3 are connected in parallel, as shown in Figure 1-1, their effective resistance is the same as a single resistance whose value R123 is given by this formula:

$$R123 = \frac{1}{\frac{1}{R1} + \frac{1}{R2} + \frac{1}{R3}}$$

Figure 1-1 Parallel resistances



The formula gives correct results for positive resistance values between 0 (corresponding to a short circuit) and ∞ (corresponding to an open circuit) inclusive. On computers that do not allow division by zero, you would have to add tests designed to filter out the cases with resistance values of zero. (Negative values can cause trouble for this formula, regardless of the style of the arithmetic, but that reflects their troublesome nature in circuits, where they can cause instability.)

Arithmetic with Infinities usually gives reasonable results for expressions in which each independent variable appears only once.

Using IEEE Arithmetic

This section provides some example computations and describes how using IEEE arithmetic in the PowerPC Numerics environment makes programming these computations easier.

Evaluating Continued Fractions

Consider a typical continued fraction $\mathrm{cf}(x)$.

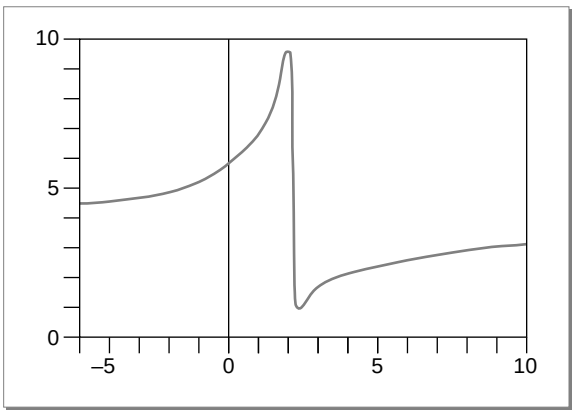
$$\mathrm{cf}(x) = 4 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}}$$

An algebraically equivalent expression is $\mathrm{rf}(x)$:

$$\mathrm{rf}(x) = \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))}$$

Both expressions represent the same rational function, one whose graph is smooth and unexceptional, as shown in Figure 1-2.

Figure 1-2 Graph of continued fraction functions $\mathrm{cf}(x)$ and $\mathrm{rf}(x)$



Although the two functions $\mathrm{rf}(x)$ and $\mathrm{cf}(x)$ are equal, they are not computationally equivalent. For instance, consider $\mathrm{rf}(x)$ at the following values of x:

$x = 1$ $\mathrm{rf}(1) = 7$

$x = 2$ $\mathrm{rf}(2) = 4$

$x = 3$ $\mathrm{rf}(3) = 8/5$

$x = 4$ $\mathrm{rf}(4) = 5/2$

Whereas $\mathrm{rf}(x)$ is perfectly well behaved, those values of x lead to division by zero when computing $\mathrm{cf}(x)$ and cause many computers to stop. In IEEE standard arithmetic, division by zero produces an Infinity. Therefore, PowerPC Numerics has no difficulty in computing $\mathrm{cf}(x)$ for those values.

On the other hand, simply computing $\mathrm{rf}(x)$ instead of $\mathrm{cf}(x)$ can also cause problems. If the absolute value of x is so big that x^4overflows the chosen data format, then $\mathrm{cf}(x)$ approaches $\mathrm{cf}(\infty) = 4$ but computing $\mathrm{rf}(x)$ encounters $(\text{overflow})/(\text{underflow})$, which yields something else. PowerPC Numerics returns NaN for such cases; some other machines return $(\text{maximum value})/(\text{maximum value}) = 1$. Also, at arguments x between 1.6 and 2.4, the formula $\mathrm{rf}(x)$ suffers from roundoff error much more than $\mathrm{cf}(x)$ does. For those reasons, computing $\mathrm{cf}(x)$ is preferable to computing $\mathrm{rf}(x)$ if division by zero works the way it does in PowerPC Numerics, that is, if it produces Infinity instead of stopping computation.

In general, division by zero is an exceptional event not merely because it is rare but because different applications require different consequences. If you are not satisfied with the consequences supplied by the default PowerPC Numerics environment, you can choose other consequences by making the program test for NaNs and Infinities (or for the flags that signal their creation).

Rather than sprinkle tests throughout the program in an attempt to keep exceptions from occurring, you might prefer to put one or two tests near the end of the code to detect the (rare) occurrence of an exception and modify the results appropriately. That is more economical than testing every divisor for zero (since zero divisors are rare).

Computing the Area of a Triangle

Here is a familiar and straightforward task that fails when subtraction is aberrant: Compute the area $A(x, y, z)$ of a triangle given the lengths x, y, z of its sides. The formula given here performs this calculation almost as accurately as its individual floating-point operations are performed by the computer it runs on, provided the computer does not drop digits prematurely during subtraction. The formula works correctly, and provably so, on a wide range of machines, including all implementations of PowerPC Numerics.

The classical formula, attributed to Heron of Alexandria, is

$$A(x, y, z) = \sqrt{s(s-x)(s-y)(s-z)}$$

where $s = (x + y + z)/2$.

For needle-shaped triangles, that formula gives incorrect results on computers *even when every arithmetic operation is correctly rounded*. For example, Table 1-2 shows an extreme case with results rounded to five decimal digits. With the values shown, rounded $(x + (y + z))/2$ must give either 100.01 or 100.02. Substituting those values for s in Heron's formula yields either 0.0 or 1.5813 instead of the correct value 1.000025.

Evidently, Heron's formula would be a very bad way for computers to calculate ratios of areas of nearly congruent needle-shaped triangles.

Table 1-2 Area using Heron's formula

	Correct	Rounding downward	Rounding upward
x	100.01	100.01	100.01
y	99.995	99.995	99.995
z	0.025	0.025	0.025
$(x+(y+z))/2$	100.015	100.01	100.02
A	1.000025	0.0000	1.5813

A good procedure, numerically stable on machines that do not truncate prematurely during subtraction (such as machines that use IEEE arithmetic), is the following:

1. Sort x, y, z so that $x \geq y \geq z$.
2. Test for $z \geq x - y$ to see whether the triangle exists.
3. Compute A by the formula

$$A = \sqrt{((x + (y + z))(z - (x - y))(x + (y - z)))/4}$$

▲ **WARNING**
This formula works correctly only if you do not remove any of the parentheses. ▲

The success of the formula depends upon the following easily proved theorem:

THEOREM *If p and q are represented exactly in the same conventional floating-point format, and if* $1/2 \leq p/q \leq 2$, *then* $p - q$ *too is representable exactly in the same format (unless* $p - q$ *suffers underflow, something that cannot happen in IEEE arithmetic).*

The theorem merely confirms that subtraction is exact when massive cancellation occurs. That is why each factor inside the square root expression is computed correctly to within a unit or two in its last digit kept, and A is not much worse, on computers that subtract the way PowerPC Numerics does. On machines that flush tiny results to zero, this formula for A fails because $(p - q)$ can underflow.

About the FPCE Technical Report

Even though many computers now conform to the IEEE standard, the standard has suffered from a lack of high-level portability. The reason is that the standard does not define bindings to high-level languages; it only defines a programming environment. For instance, the standard defines data formats that should be supported but does not tell how these data formats should map to variable types in high-level languages. It also specifies that the user must be able to control rounding direction but falls short of defining how the user is able to do so.

However, the definition of a binding is in progress for the C programming language. The Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group (NCEG), or **ANSI X3J11.1,** has proposed a general floating-point specification for the C programming language, called the **FPCE technical report,** that contains additional specifications for implementations that comply with IEEE floating-point standards 754 and 854.

The FPCE technical report not only specifies how to implement the requirements of the IEEE standards, but also requires some additional functions, called **transcendental functions** (sometimes called *elementary functions*). These functions are consistent with the IEEE standard and can be used as building blocks in numerical functions. The transcendental functions include the usual logarithmic and exponential functions, as well as $\ln(1 + x)$ and $e^x - 1$; financial functions for compound interest and annuity

calculations; trigonometric functions; error and gamma functions; and a random number generator. The **PowerPC Numerics library,** contained in the file MathLib, implements the transcendental functions.

Part 2 of this book describes how PowerPC Numerics complies with the recommendations in the FPCE technical report.

PowerPC Numerics Versus SANE

Although PowerPC Numerics is an implementation of the IEEE Standard, it is not the **Standard Apple Numerics Environment (SANE).** SANE is the numerics environment used on **680x0-based Macintosh computers,** and it is the numerics environment used when you run a 680x0 application on a PowerPC processor-based Macintosh computer. PowerPC Numerics is the environment used when you run an application built for a PowerPC processor-based Macintosh computer.

There are fundamental differences between PowerPC Numerics and SANE because of the differences in the microprocessors on which the two environments are used. The major difference is that SANE supports an 80-bit extended type and performs all floating-point computations in extended precision. This protects the user from roundoff error, overflows, and underflows that might occur in an intermediate value when determining the result of an expression. Because the PowerPC processor is double-based, support of an 80-bit data type would be inefficient. It instead supports a 128-bit type (in software) called *double-double* (which corresponds to the `long double` type in C). PowerPC Numerics provides this wide type only for cases where precision greater than that provided by the double format is necessary; PowerPC Numerics does not perform all computations in double-double precision. Instead, PowerPC Numerics recommends a method by which an expression is evaluated in the widest precision necessary (see Chapter 3, “Expression Evaluation”).

Another fundamental difference is that PowerPC Numerics conforms to the FPCE recommendations as well as to the IEEE standard. C implementations using SANE do not necessarily comply with the FPCE recommendations.

See Appendix A, “SANE Versus PowerPC Numerics,” for more information on the differences between PowerPC Numerics and SANE.