Chapter 2

Computer representation of numbers

2.1 Introduction

Before delving into code development, we first need to think about the basic ingredient that lies at the core of all scientific computations: numbers. Computer languages generally provide specific ways of representing fundamental or *primitive* data types that correspond to numbers in some direct manner. For C++ these primitive data types include

- *boolean* (or `bool`) for variables that take on the two values `true` and `false` or, equivalently, the values 1 (= `true`) and 0 (= `false`),
- *integer*, which translates into specific types such as `short`, `unsigned short`, `int`, `long int` and `unsigned long int` that provide different amounts of storage for integer values,
- *character*, indicated by the modifier `char` in source code, that is used for variables that have character values such as "A" and
- *floating point* which encompasses basically all non-integer numbers with the three storage types `float`, `double` and `long double`.

Ultimately, all computer operations, no matter how sophisticated, reduce to working with representations of numbers that must be stored appropriately for use by a computer's central processing units or CPUs. This process is accomplished through the manipulation of transistors (in CPUs and random access memory or RAM) that have two possible states: off and on or, equivalently, 0 (= off) and 1 (= on). By combining transistors it becomes possible to create a dyadic representation for a number that allows it to be stored and used in arithmetic operations. Of course, the number of transistors is finite which affects both how and how much information can actually be stored. To manage overall memory (i.e., storage) levels effectively it is necessary to restrict the amount of memory that can be allocated to different kinds of numbers with the consequence that there are limits on how big or small a number can be and still be stored in some meaningful fashion. To appreciate the implications of this we first need to think a bit about computer arithmetic.

In the familiar decimal (or base 10) system, numerical values are represented in units or powers of 10. For simplicity, let us work with only nonnegative integers for the moment. Then, the basis representation theorem from number theory (e.g., Andrews 1971) has the consequence that any such integer, k, can be written as

$$k = \sum_{j=0}^{m} a_j (10)^j \tag{2.1}$$

for some unique (if we require $a_m \neq 0$) integer m and some unique set of integer coefficients $\{a_0, \ldots, a_m\}$. As an example,

$$193 = 1 * (10)^2 + 9 * (10)^1 + 3 * (10)^0$$

with `*` indicating multiplication. So, in this case $m = 2, a_0 = 3, a_1 = 9$ and $a_2 = 1$. Note that for the coefficients in (2.1) to be unique their range must be restricted to, e.g., $\{0, \ldots, 9\}$.

Since transistors have only two states it is not surprising that the base of choice for computer arithmetic is binary or base 2. The base 2 version of (2.1) is the *binary expansion*

$$k = \sum_{j=0}^{m} b_j (2)^j \tag{2.2}$$

with m a nonnegative integer for which $b_m \neq 0$. The b_j's are coefficients that can take on only the values 0 or 1. Returning to our example, it can be seen that

$$193 = 1 * (2)^7 + 1 * (2)^6 + 1 * (2)^0.$$

This corresponds to (2.2) with $m = 7$, $b_7 = b_6 = b_0 = 1$ and $b_5 = \cdots = b_1 = 0$.

The fact that the coefficients of a number in binary form are all zeros and ones means that it can be represented as a sequence of zeros and ones with the zeros and ones corresponding to the b_j in (2.2) and their location in the sequence indicating the associated power of two for the coefficient. The slots that hold the coefficients in a sequence are called *bits* with a sequence of eight bits being termed a *byte*. For the number 193 this translates into a tabular representation such as

power of two	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
coefficient	1	1	0	0	0	0	0	1

That is, in binary 193 can be represented exactly in one byte as the value 11000001.

The connection between machine memory and binary arithmetic becomes complete once bits and bytes are identified with individual transistors and blocks of eight transistors. To represent a number in memory its binary representation is physically created by allocating it a block of memory (e.g., a group of contiguous transistors), identifying the individual transistors in the block with a power of 2 from its binary representation and then turning on those transistors that correspond to powers of 2 that have unit coefficients. To belabor the point a bit, a block of eight transistors would be needed to hold the integer 193; the first, seventh and eighth transistor in the block would be turned on and the other five would be turned off. Note that it will take all eight bits/transistors to store 193 and any fewer would not be capable of doing the job.

Just as 193 cannot be stored in anything less than eight bits there is a limit to the size of numbers that can be stored in any finite number of bits. For example, the largest number that can be stored in one byte is

$$\sum_{j=0}^{7} (2)^j = 2^7 \sum_{j=0}^{7} 2^{-j} = 2^7 \frac{1-(1/2)^8}{1-(1/2)} = 2^8 - 1 = 255. \tag{2.3}$$

More generally, the largest number that can be stored in m bits is $2^m - 1$ (Exercise 2.1).

2.2 Storage in C++

Let us now consider how the ideas about number representation from the previous section are implemented in C++. While investigating this issue we will also introduce some of the language features that will allow us to begin to write, compile and execute C++ programs.

The amount of storage that is allocated to different data types can be determined using the `sizeof` operator that C++ inherits from C. Listing 2.1 below demonstrates the use of this operator to determine information about storage space, in bytes, that is provided for some of the primitive data types discussed in the previous section.

Listing 2.1 *storageSize.cpp*

```
//storageSize.cpp
#include <iostream>

using namespace std;

int main()
{
  cout << "The storage allocated for a char is " << sizeof(char)
       << " bytes" << endl;
  cout << "The storage allocated for an unsigned short integer is "
       << sizeof(unsigned short int) << " bytes" << endl;
  cout << "The storage allocated for an integer is " << sizeof(int)
       << " bytes" << endl;
  cout << "The storage allocated for a long integer is "
       << sizeof(long int)  << " bytes" << endl;
  cout << "The storage allocated for an unsigned long integer is "
       << sizeof(unsigned long int) << " bytes" << endl;
  cout << "The storage allocated for a float is " << sizeof(float)
       << " bytes" << endl;
  cout << "The storage allocated for a double is " << sizeof(double)
       << " bytes" << endl;
  cout << "The storage allocated for a long double is "
       << sizeof(long double) << " bytes" << endl;
  return 0;
}
```

Now this is a very simple program. But, it illustrates some of the basic syntax that will be seen again and again. At the outset there is a *comment statement* (signified by `//`) that gives us the name of the program file, a practice we will adhere to throughout the book. The use of two double forward slashes causes the compiler to ignore the succeeding text on a line. Had we wanted a comment that ran more than one line this could have been accomplished using either `//` at the start of each line or by bracketing the comment encompassing possibly multiple lines by `/*` and `*/`.

The comment in Listing 2.1 is followed by the statement `#include <iostream>` which is a directive to the preprocessor to include information about the input/output classes in the C++ Standard Library. The actual code for the library has already been compiled and will automatically be linked with our program during the compilation process. There are many other tools available from this library beyond just those for input and output. They can be accessed similarly using include directives and the specific files that are included are called *header* files. Such files will be discussed in more detail in the next chapter. But, as one possible example, the use of `#include <cmath>` provides access to many of the standard mathematical functions such as the logarithm, trigonometric functions, etc.

The next statement in the code, `using namespace std;`, ensures that there will be no ambiguity in referring to functions and classes that have any of the specific names that have been given to those provided in the C++ Standard Library. Specifically, Listing 2.1 uses a `cout` object and the output manipulator `endl` that are declared in the iostream header file. The names of these operators are made globally available with the `using namespace std;` statement. Omission of this line from our code would have caused the compiler to generate an error message such as

```
storageSize.cpp: In function ‘int main()’:
storageSize.cpp:6: error: ‘cout’ was not declared in this scope
storageSize.cpp:7: error: ‘endl’ was not declared in this scope
```

Basically, a *namespace* is a collection of definitions of variables, functions and other key components associated with a library or program that have been gathered together for various possible reasons. Among other things namespaces provide a mechanism for reusing desirable names for functions and classes that may have already been employed in some other context. It is possible to bypass the `using` directive via use of the *scope resolution operator* `::` as discussed in Section 3.8. For now we will travel the simpler path of stating up front that all references are to the Standard Library.

We now come to the start of the program signaled by the appearance of the word `main`. Every C++ program must have one and only one `main` function. It serves the purpose of directing the flow of activity within a program. While there may be many functions and objects whose actions are intricately interlaced throughout a program, whatever takes place must have been initiated by directions from `main`. The body of `main` (as well as that of any other function) is enclosed by the matching curly braces { and }.

In terms of content, `main` in Listing 2.1 is just a collection of explanatory text to be printed with values that are returned by the `sizeof` operator. An application of `sizeof` to a particular data type (supplied as its argument) returns the number of bytes that are used for its storage.

The symbols `<<` and `>>` are the output and input insertion operators that work with the standard output and input stream objects `cin` and `cout`, respectively, that relay information to and from (typically) a shell. The insertion operators provide the facility to chain together input/output operations. The output *manipulator* `endl` that appears at the end of each output line produces a carriage return (or starts a new line) and flushes (or cleans out) the output buffer. You will see the effect of all this when we look at some output from the program. Note that each full line of code in `main` is ended with a semicolon.

The final line of the program is `return 0;`. In general the `return` statement has the effect that one might expect; it returns control to a calling function. In the case of how it is used here in `main`, it transfers control back to the operating system. An exit integer with a value of 0 is returned upon completion of the program. The fact that 0 is viewed as an integer is a consequence of the `int` designation that immediately precedes `main`. This is true more generally in that a function must return a value of the same type that appears in its definition with the exception of `main` functions and functions with return type `void`. The `return` statement could actually have been omitted from Listing 2.1 without causing compilation errors. The `void` return type will be discussed shortly.

The "storage size" source code was saved as storageSize.cpp. The cpp file extension is one of the permissible options for the GNU C++ compiler that is available in Unix environments. The others are .cc, .cxx and .C. To run the program, we first compile it using

```
$ g++ -Wall storageSize.cpp -o storageSize
```

The compilation command begins with g++ that invokes the GNU compiler. This has the effect of transforming the input file storageSize.cpp into machine language. In doing this, there were two compiler options that were employed: -Wall turns on all of the most commonly used compiler warnings while -o allows us to specify the name of the executable, as storageSize in this instance. Without the -o option the executable would be named a.out by default; the a.out name is an abbreviation of "assembler output".

To load and run the compiled program the name of the executable (i.e., storageSize) is entered on the command line prefaced by the ./ modifier that informs the shell where to look for the executable. In this particular instance, the result is

```
$ ./storageSize
The storage allocated for a char is 1 bytes
The storage allocated for an unsigned short integer is 2 bytes
The storage allocated for an integer is 4 bytes
The storage allocated for a long integer is 8 bytes
The storage allocated for an unsigned long integer is 8 bytes
The storage allocated for a float is 4 bytes
The storage allocated for a double is 8 bytes
The storage allocated for a long double is 16 bytes
```

The results obtained from the storageSize program will be machine-dependent. For the C++ implementation on the computer that was used to produce this output, a small integer with no attached sign as represented in `unsigned short` will be stored in two bytes or 16 bits of memory. Both the `long int` and `unsigned long int` storage types provide eight bytes of storage while the values for an `int` variable can take up no more than four bytes of memory.

One of the reasons for machine dependence is the flexibility that was written into the C++ Standard. Only minimum and maximum values are specified there for the different data types. For example, in the case of integers the requirement is that `long int` need only provide as much storage as `int`. The values that we just saw certainly satisfy this guideline since `long int` provides twice the storage capacity of `int`.

The `float` and `double` types are the irrational number analog of the `int` and `long int` storage types for integers. Another phrase that is used for `float` is *single precision* and `double` is short for *double precision*. This terminology becomes meaningful from the output of the storageSize program; a `double` is allocated 64 bits of memory or twice that which is allowed for a `float`. A `long double` is then stored in 128 bits of memory or twice that provided for a `double`.

2.3 Integers

Computer arithmetic is basically conducted with integers and floating-point values that are used to approximate the values of non-integers. Although floating-point storage has a more complex structure, it is not entirely dissimilar from what is employed for integers. In the end every number must be translated into a sequence of bits and the devil is in the details of how this is done. Integers are a good place to begin to think about such concepts and, accordingly, that is where this section begins.

To explore the way integers are stored in some specific cases of interest it will be helpful to have a program that returns the binary representation of a number. To accomplish this let us first deal with the `unsigned short int` or simply `unsigned short` data type. As the name suggests, these are small nonnegative integers. As seen in the previous section, they occupy 2 bytes of storage with values ranging from 0 to $2^{16} - 1 = 65535$. This makes them ideal for our current purpose where we want to extract the binary representation of numbers in a simple setting. A program that accomplishes this is provided by Listing 2.2.

Listing 2.2 *binaryRep.cpp*

```
//binaryRep.cpp
#include <iostream>

using namespace std;

void printBinary(unsigned short val);

int main(){
  unsigned short inVal;
  cout << "Enter a number between 0 and 65535: ";
  cin >> inVal;
  cout << "Your number in binary is ";
  printBinary(inVal);
  cout << endl;
  return 0;
}

void printBinary(unsigned short val){
  for(int i = 15; i >= 0; i--)
    if(val & (1 << i))
      cout << "1";
    else
      cout << "0";
}
```

Listing 2.2 employs some elementary bit-wise operations in a function called `printBinary` to pick off the internal binary representation of an integer. The first thing we see in the listing is a *prototype* or *declaration* of the basic form for `printBinary` that appears prior to the function `main`. The *return type* of the function (i.e., the keyword that immediately precedes the function's name) is designated as `void` which means it does not return a value to the calling program. In C/C++ it is necessary to tell the compiler about the essential details (e.g., return type and type for the arguments) in a function before it can be used. Because `printBinary` is to be used in the function `main`, we need to either define it prior to `main` or furnish a prototype or sketch of the function prior to calling it in `main`. The second option was chosen here. But, the program would have compiled and produced the same results if the entire definition of `printBinary` was moved from the bottom of the listing and placed before `main`.

The first line in `main` concerns the variable `inVal` that will hold the integer whose binary representation will be determined. The line consists of a *type declaration* which states that `inVal` is of storage type `unsigned short`. In C++ every variable's type must be explicitly stated when it is introduced into the program. Failure to do so will produce a compilation error. Indeed, omission of the `unsigned short inVal;` line from our program would have generated a message such as

```
binaryRep.cpp: In function   int main()   :
binaryRep.cpp:10: error: 'inVal' was not declared in this scope
```

from the compiler.

The `main` function in Listing 2.2 also illustrates the use of `cout`'s input partner `cin` to read a value for `inVal` from the shell command line. The `printBinary` function is then applied to `inVal` with the results being written to the shell before the program terminates.

We now arrive at the actual definition of the `printBinary` function. It takes as input an `unsigned short` integer that is two bytes in size. Such numbers can be viewed as having 16 possible slots corresponding to 2^0 all the way to 2^{15}. The program then steps through each of these slots using a `for` loop that begins with a look at slot 16 (i.e., the 2^{15} slot) and works backward to the first (or 2^0) slot.

The C++ version of a `do` or `for` loop consists of specifying three quantities. First, a starting value is given for the index variable whose value governs the iteration. In this case the index variable is `i` whose type is first designated as `int` before setting it to 15. Then, there is logical condition (i.e., `i >= 0`) that terminates the loop when it evaluates to `false`. Finally, direction is provided on how the loop should move through the values of the index variable. This is accomplished here with the `i--` syntax which says that `i` should be decremented by one at each step through the loop. Section 3.4 provides more details on the use of loops in C++.

The `printBinary` function picks off the binary representation of the `unsigned int` variable supplied for its argument by first shifting the integer 1 ($= 2^0$) over `i = 15` slots using the `1 << 15` operation. We have already seen the symbol `<<` used in a very different context in conjunction with `cout`. The question this raises is how one operator can be used for two completely different purposes. The answer is that this is an example of polymorphism accomplished via a feature called operator overloading that will be developed in more detail in the next chapter. By using this facility it becomes possible to have operators that adjust automatically to the situation where they are applied.

Basically, the operation for `i = 15` at the beginning of the `for` loop in Listing 2.2 multiplies 1 (or 0000000000000001) by 2^{15} to obtain 1000000000000000. The `if` statement then relies on a conditional comparison of `val` to 1000000000000000 using the bit-wise AND operator `&`. Here `&` takes the binary representation of two numbers as input and returns a binary representation that has ones in all the places where *both* numbers have ones and zeros elsewhere. So, in terms of the way we have set this up, the argument in the `if` statement (i.e., `val & (1 << 15)`) will evaluate to zero (or `false` in terms of how it is viewed by the `if` conditional) unless `val` has a 2^{15} term in its binary representation. In the latter case `val & (1 << 15)` evaluates as `true` causing a 1 to be written to the shell with a 0 being written otherwise. This same process proceeds across descending powers of 2 as a result of the decrement operator `i--` that reduces the value of `i` by one after the comparison in the `if` statement is executed.

After compilation using

```
$ g++ -Wall binaryRep.cpp -o binaryRep
```

the program will produce results like

```
$ ./binaryRep
Enter a number between 0 and 65535: 2
Your number in binary is 0000000000000010
$ ./binaryRep
Enter a number between 0 and 65535: 65535
Your number in binary is 1111111111111111
```

The output suggests that the code is working as expected.

Up to this point, only nonnegative integers have been considered. To distinguish between negative and positive integers, one must retain information about the sign. For example, we could just let the last bit correspond to the sign. This is called the *signed magnitude* approach. To illustrate the idea, suppose for simplicity we are working with a one byte or eight-bit integer. Then, for example, the binary representation for 12 will be 00001100 and -12 would be represented by 10001100. By reserving the last bit for the sign we have left ourselves with only 7 "working" bits. Accordingly, the numbers that can be represented fall between $-(2^7-1) = -127$ and $2^7-1 = 127$.

Another way to store signed integers is the *two's complement* rule. Here the left-most bit is allowed to represent -2^{m-1} in an m bit storage allocation. For the case of $m = 8$ in our example, the last bit corresponds to $-2^7 = -128$. One can work backward from this to get other negative integers by allowing the other seven (more generally, $m-1$) bits to provide coefficients for positive powers of two as before. So, with a slight abuse of notation, we can write

$$-12 = -128 + 64 + 32 + 16 + 4 = 11110100.$$

The two's complement approach with $m = 8$ allows us to represent all the negative integers between -128 and $-1(= -128 + (2^7-1))$ and, hence, the integers that can be represented are now all those in the range -128 to 127. That is, one additional integer has been gained over the signed magnitude method. This result extends directly to the general case of m bits with the consequence that under the two's complement approach it is possible to represent all integers between

$$\overbrace{100\cdots 0}^{m\,\text{times}} = -2^{m-1} \tag{2.4}$$

and

$$\overbrace{0111\cdots 1}^{m\,\text{times}} = 2^{m-1} - 1. \tag{2.5}$$

To adapt Listing 2.2 to handle signed integers it is only necessary to drop the `unsigned` designation from `inVal` and `val` in Listing 2.2. The allowable input values for a two's complement storage scheme will be $2^{15} = -32768$ to $2^{15} - 1 = 32767$. Using the corresponding modified version of Listing 2.2, the output below allows us to deduce that the computer on which this code was compiled and executed uses the two's complement rule to represent `int` variables.

```
$ ./binaryRep
Enter a number between -32768 and 32767: -32768
Your number in binary is 1000000000000000
$ ./binaryRep
Enter a number between -32768 and 32767: 32767
Your number in binary is 0111111111111111
```

2.4 Floating-point representation

Storage types with fixed positional representations work fine for integers. This basic idea could even be used, at least in principle, to represent rational numbers or fractions as they can be expressed as k/m with k and m both integers. However, the utility of the fixed point framework ends here and leaves unresolved the problem of storing and working with irrational numbers, which comprise the bulk of the real number system.

At this point it must be realized that a general real number cannot be stored in its entirety and, as a result, in most cases the stored value will represent only an approximation to the truth. Errors are created in computer arithmetic with real numbers due to both the rounding of the numbers for storage as well as further manipulations. These issues will be discussed in the next section. For the present it suffices to recognize that there is a limit to the precision that can be achieved from any computer representation that might be employed for irrational numbers. We will express the precision by the number of significant digits of agreement between the true value of a number and its floating-point representation. A good storage system is one that attempts to minimize losses in precision subject to the constraints that have been imposed on the allowed amount of storage.

Roughly speaking, significant digits are obtained by removing leading and trailing zeros from a number as would take place for conversion to scientific notation, for example. This is, in fact, an apt analogy for the developments in this and the next section. To effectively store irrational numbers on a computer the decimal point must be allowed to float. By this we mean that the decimal is always placed after the number's first significant digit. To recover the actual value of the number this decimal relocation is accompanied by multiplication by the base (e.g., 2 or 10) raised to an appropriate power. The result is an adaptive representation for numbers that allows for considerable storage flexibility relative to the integer case.

To be somewhat more specific, let us return to the base 10 scenario where the floating decimal concept is already familiar in the form of scientific E-notation. For example, under this format the values 21.237 and .021237 would be written as 2.1237E+1 and 2.1237E−2. The use of "E" in this context stands for the exponent of 10 and all that is really being stated is that $21.237 = 2.1237 \times 10^1$ and $.021237 = 2.1237 \times 10^{-2}$. This idea works quite generally in that $m + 1$ significant figures can be retained for a real number x by writing it as

$$x \doteq \pm a_0.a_1a_2\cdots a_m 10^p, \tag{2.6}$$

where $\doteq$ indicates approximate equality, $a_0, \ldots, a_m$ are all integers between 0 and 9 with $a_0 \neq 0$ and p is a signed integer.

The idea behind (2.6) is that any real number can be written as

$$x = \pm \sum_{j=0}^{\infty} a_j 10^{p-j} \tag{2.7}$$

and (2.6) comes from retaining only $m+1$ terms in the series (2.7). Since we only want to include significant digits in the approximation to x, cases where $a_0 = 0$ can be excluded. Thus, subject to a rounding rule that determines a_m (e.g., augment a_m by one if $a_{m+1} \geq 5$), the approximation (2.6) is uniquely determined. In the case of 21.237, $m = 4$ and $p = 1$ while $m = 4$ and $p = -2$ for .021237.

There is nothing special about base 10 and the analog of (2.6) for base 2 looks like

$$x \doteq \pm b_0.b_1b_2 \cdots b_m 2^p \tag{2.8}$$

with $b_0, \ldots, b_m$ all having values of either 0 or 1 and p, again, being a signed integer. As was the case for (2.6), (2.8) derives from writing

$$x = \pm \sum_{j=0}^{\infty} b_j 2^{p-j}$$

and then keeping the leading $m+1$ terms from the series. It is only necessary to deal with cases where $b_0 = 1$ which means that a binary representation can always be written in the form

$$x \doteq \pm 1.b_1b_2 \cdots b_m 2^p. \tag{2.9}$$

This has the practical consequence that the value of b_0 need not actually be stored. Numbers represented as in (2.9) are said to have been *normalized.* The remaining part of the approximation $.b_1b_2 \cdots b_m$ is called the *mantissa* or *significand.*

Let us now see how an expression such as (2.9) might be translated into a specific storage scheme. We will begin with a simple, nonrealistic, illustration and then describe what can be expected on a typical machine.

Suppose now that a number x is to be stored in a floating-point representation that is allocated one byte of storage. There are essentially four things that must be accounted for: the sign of the number, the value of the exponent for 2 in (2.9) as well as its sign and, finally, the values for $b_1, \ldots, b_m$. To account for the sign for x, take the left-most bit to be a sign bit with value 0 for a positive number and 1 for a negative value. The next three bits can then be used to hold a two's complement representation of the exponent (that can range from -4 to 3). This leaves four bits (i.e., $m = 4$) to hold the significand. To apply this idea to, e.g., the number -12.5, first observe that

$$\begin{aligned} 12.5 &= 2^3 + 2^2 + 2^{-1} \\ &= 1.1001 \times 2^3. \end{aligned}$$

Hence, -12.5 would be represented as

$$\underbrace{1}_{\text{for the }-} \quad \underbrace{011}_{\text{for 3}} \quad \underbrace{1001}_{\text{for the significand}}$$

The leading 1 in the significand has been dropped because it is assumed *a priori* to be 1, and the remaining part, 1001, that is stored represents coefficients for $2^{-1}, 2^{-2}, 2^{-3}$ and 2^{-4}, respectively.

Now consider adding .25 to -12.5. We can represent .25 (exactly) in our storage plan with

$$\begin{aligned} .25 &= 2^{-2} \\ &= 1.0000 \times 2^{-2}. \end{aligned}$$

But, to add it to -12.5 a common denominator is needed which, in this case, entails viewing .25 as

$$.25 = .00001 \times 2^3.$$

However, in this form it cannot be stored exactly in four bits and instead must be rounded to a significand of .0000 if we chop off the last digit or to .0001 if we round up. As a result, addition returns the sum of -12.5 and .25 as -12.5 under truncation and -12 if the answer is rounded up. This is an illustration of the types of problems that can occur when performing basic arithmetic operations using numbers that have very different magnitudes. In fact, in this case the addition of a number a to -12.5 leaves it unchanged for any $|a| \leq 2^{-3}$.

Finally, let us see what happens when we try to store 1/5 using our simple system. This fraction cannot be stored exactly and, instead, the best that can be obtained is an approximate representation based on the fact that

$$\begin{aligned} \frac{1}{5} &\doteq 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} \\ &= .19921875. \end{aligned}$$

The sign bit for .19921875 will have value 0 and the two's complement representation for its exponent for 2 (namely, -3) is 101. The significand cannot be stored in its entirety and rounding is required to make it fit into four bits of memory. The value is rounded up to 1010 and that is what is retained. The final result is that 1/5 receives the floating-point representation

$$\underbrace{0}_{\text{for the -}} \quad \underbrace{1001}_{\text{for the exponent}} \quad \underbrace{1010}_{\text{for the significand}}$$

i.e., we approximate 1/5 by $2^{-3} + 2^{-4} + 2^{-6} = .203125$.

There is a slight problem with our simple storage scheme. Since the lead bit of the significand was not stored the natural way to represent 0 has been lost. To account for this 0 can simply be represented by some value such as `01000000` which would have the effect of eliminating 2^{-4} as a possible value.

In the real world storage for floating-point numbers can be expected to abide by the IEEE 754 standard discussed, for example, in Stevenson (1981) and Goldberg (1991). In the case of, e.g., `float` types the 754 standard specifies four bytes of storage. The first of the 32 available bits is a sign bit with the next eight bits being allocated for storage of the exponent. The significand is then stored in the remaining 23 bits.

In contrast to the way the exponent was handled in our simple storage scheme, the 754 standard employs a *biased* exponent. With eight bits there are only 256 possible integers (e.g., 0 to 255 or -128 to 127) that can be stored whether they are signed or unsigned. Biasing then works with the unsigned integers 0 to 255 that can be stored in eight bits and transforms or biases the values by subtracting off 127 to produce exponents (in base 2) that range from $-127(= 0 - 127)$ to $128 = (255 - 127)$.

To illustrate the IEEE storage scheme, consider the number -193.625 as a four byte `float`. Now, note that

$$193.625 = 1 * (2)^7 + 1 * (2)^6 + 1 * (2)^0 + 1 * (2)^{-1} + 1 * (2)^{-3}$$

which means that the value in binary is

$$11000001.101 = 1.1000001101 * 2^7.$$

The first bit in the floating-point representation for -193.625 will be 1 to indicate that the value is negative. The next eight bits represent the exponent: i.e., 7. Instead of 7, the biasing approach leads to storage of

$$\begin{aligned} 134 &= 7 + 127 \\ &= 2^7 + 2^2 + 2^1 \end{aligned}$$

which is the binary sequence 10000110. The final result is the normalized representation

$$\underbrace{1}_{\text{for the } -}\ \underbrace{10000110}_{\text{for } p}\underbrace{10000011010000000000000}_{\text{for the significand}}$$

As was true for our simple storage scheme, normalization entails the loss of the natural choice for 0. The problem is resolved by defining 0 as 2^{-127}: i.e., 0 is

$$\underbrace{0}_{\text{sign}}\ \underbrace{00000000}_{\text{exponent}}\underbrace{00000000000000000000000}_{\text{significand}}$$

which translates to 2^{-127} after biasing of the exponent.

Unlike the integer case, extracting the binary representation of a floating-point number is a little tricky. This is due to the fact that the bit-wise operators used in the integer case cannot be used directly with floating-point numbers. A standard recommendation for an end run around this difficulty looks something like the code in the listing below.

```
#include <iostream>

using namespace std;

void printbinary(char val){
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      cout << "1";
    else
      cout << "0";
}

int main(){
  float f;
  cout << "enter a number: ";
  cin >> f;
  char* pf = reinterpret_cast<char*>(&f);
  cout << "your number in binary is ";
  for(int i = sizeof(float) - 1; i >= 0; i--)  printbinary(pf[i]);
  cout << endl;
  return 0;
}
```

This program illustrates two important aspects of C/C++ that will become essential for later chapters: namely, pointers and addresses. These quantities will be discussed in some detail in the next chapter. For now let us merely mention their general purpose and how they are used in this particular program.

The syntax `char* pf` in the program says that `pf` is a pointer to `char`. This means it is a variable whose value is the address in memory that is occupied by a `char` variable. The address that has been assigned to `pf` is essentially that of the floating-point variable `f` which is obtained through the syntax `&f`. The problem is that `&f` is the address of a `float` while `pf` is expecting to receive the address of a `char` for its value. To make the transition from a `float` to a `char` address, a *cast* is used. In general, casting is the process of changing a variable from one data type into another in terms of how it is viewed by the compiler. C++ performs implicit casting in standard cases such as transforming an integer into a floating-point value and similar types of conversions. More generally, an explicit cast of a variable `x` to a new data type `newType` is accomplished with syntax of the form

```
  newType y = (newType)x;
```

provided the conversion is possible. For example, the ensuing code segment has the effect of creating a variable of type `double` from one of type `int`.

```
  int x = 1;
  double y = (double)x;
```

Our particular problem goes beyond the capability of a simple explicit cast such as this. We need to transform the memory address of a `float` into one that is treated as holding a `char`. For this purpose it is necessary to use the `reinterpret_cast<char*>` operator. This basically tells the compiler to forget the original type of data that was stored at `&f` and allow this location in memory to be viewed as memory for variables of type `char`. Our storageSize program revealed that variables of type `char` are allocated 1 byte of memory. Thus, after the cast the memory located at `&f` will be viewed by `pf` as representing the starting point for `sizeof(float)` (or, 4 in this instance) one-byte blocks of memory. The resulting pattern of on/off positions for the sequence of transistors found at these locations can be interpreted in a variety of ways. The goal is to figure out which bits are set to 1 and which are set to 0. This can be done by applying our bit-wise shift operator, that works equally well with `char` variables, on each one-byte block of memory. The new `main` function also illustrates the dereferencing of the `pf` pointer when the argument `pf[i]` is passed to the `printBinary` function. This is a vector-type indexing property of pointers wherein syntax such as `pf[i]` gives access to the contents of the `i`th block of memory rather than the value of the pointer itself.

An example of output from the floating-point representation program is

```
Enter a number: -193.625
Your number in binary is 11000011010000011010000000000000
```

The binary representation agrees with what was done before which implies that the IEEE standard for `float` is being employed on the machine that ran the program.

To conclude this section let us briefly discuss the mechanics of addition/subtraction and multiplication of floating-point numbers. First consider the multiplication of x_1 times x_2 with

$$x_i = 1.b_{i1} \cdots b_{im} 2^{p_i} \tag{2.10}$$

for $i = 1, 2$. To compute the product $x_1 x_2$ exactly there are 2 steps: namely, 1) add the two exponents and 2) multiply the two significands $1.b_{11} \cdots b_{1m}$ and $1.b_{21} \cdots b_{2m}$. This latter step results in a $2m + 2$ binary decimal number of the form $1.c_1 \cdots c_{2m+1}$. The answer is then

$$x_1 x_2 = 1.c_1 \cdots c_{2m+1} 2^{p_1+p_2}.$$

Of course, if only m decimal bits are available for storage, this answer must be rounded and even the evaluation of the coefficient c_{m+1} that would be needed to accomplish the rounding can be seen as problematic. The use of extended precision for doing the computation prior to rounding is one way to solve this problem. But, other approaches are used in practice as indicated in the next section.

Now consider adding x_1 and x_2 in (2.10). This process is more involved as a result of the common denominator that must be used to carry out the addition process as was illustrated with the addition of .25 to -12.5 under our eight-bit floating-point scheme. Assume that

$p_1 > p_2$ and then shift x_2 to have the form

$$x_2 = 0.0 \cdots 01b_{21} \cdots b_{2m}2^{p_1} \tag{2.11}$$

with $p_1 - p_2 - 1$ bits being 0 counting from the first bit on the right side of the decimal. The sum or difference can now be computed through binary addition or subtraction of the original x_1 and shifted x_2 significands with the result being rounded to have m bits to the right of the decimal. There are practical considerations that arise here in terms of how to store the shifted version of x_2 and problems arise if only m decimal bits are used to store the shifted value. Guard bits (e.g., Kaneko and Liu 1991) can be used to deal with such issues.

2.5 Errors

As seen in the last section, most numbers cannot be represented exactly in floating-point format. In this section we will explore the consequences of this for the precision of computations that are carried out on computers in binary arithmetic.

First let us address the losses that can arise from conversion of a number to its floating-point representation. It is possible to place bounds on the amount of error that can be incurred. The bounds are, of course, dependent on the type of "rounding" that is employed. The simplest approach is truncation or chopping.

Sign plays no role in this development. So, suppose that x is a positive real number that can be written as

$$x = \sum_{j=0}^{\infty} b_j 2^{p-j}$$

with $b_0 = 1$. Then, *chopping* replaces x by the approximation

$$\tilde{x} = 2^p + \sum_{j=1}^{m} b_j 2^{p-j}$$

for some integer m. The error that is incurred by this approach is

$$\begin{aligned} x - \tilde{x} &= \sum_{j=m+1}^{\infty} b_j 2^{p-j} \\ &\leq 2^p \sum_{j=m+1}^{\infty} 2^{-j} \\ &= 2^p 2^{-(m+1)} \sum_{j=0}^{\infty} 2^{-j} \\ &= 2^{p-m}. \end{aligned}$$

Instead of chopping an alternative approximation $\tilde{x}$ can be produced by rounding to the value that is closest to x in the sense of minimizing $|x - \tilde{x}|$. This cuts the bound in half to $2^{p-(m+1)}$ as we now demonstrate.

First observe that the *round-to-closest value* choice for $\tilde{x}$ is obtained by increasing b_m by one if $b_{m+1} = 1$ and using the chopped approximation otherwise. Consequently,

$$\tilde{x} = 2^p \left[\sum_{j=0}^{m} b_j 2^{-j} + \tilde{b}_{m+1} 2^{-(m+1)} \right]$$

with $\tilde{b}_{m+1}$ being either 2 or 0 depending on whether or not b_{m+1} is 1 or 0, respectively.

Thus, when $b_{m+1} = 1$ the error is

$$|x - \tilde{x}| = 2^p \left| 2^{-(m+1)} - \sum_{j=m+2}^{\infty} b_j 2^{-j} \right| \le 2^{p-(m+1)}.$$

On the other hand, if $b_{m+1} = 0$, $x - \tilde{x} \le 2^{p-(m+1)}$ from our analysis of rounding by chopping.

To translate the previous bounds to relative absolute error, observe that, since $|x| \ge 2^p$,

$$\frac{|x - \tilde{x}|}{|x|} \le 2^{-(m+1)} \tag{2.12}$$

for round-to-closest value and

$$\frac{|x - \tilde{x}|}{|x|} \le 2^{-m}$$

for chopping. These values are sometimes referred to as the *machine epsilon*. An application of (2.12) to the IEEE floating-point format gives error bounds on the order of $2^{-24} \doteq .6(10^{-7})$ for single precision and $2^{-53} \doteq 10^{-16}$ for double precision arithmetic (Exercise 2.8).

Define the relative error from floating-point approximation to be

$$E = \frac{\tilde{x} - x}{x} \tag{2.13}$$

with $\tilde{x}$ the approximation to x obtained from either one of the rounding schemes. The relative error satisfies

$$\begin{aligned} \tilde{x} &= x + \tilde{x} - x \\ &= x\left(1 + \frac{\tilde{x} - x}{x}\right) \\ &= x(1 + E) \end{aligned} \tag{2.14}$$

which gives it a simple interpretation in terms of the way it measures the disparity between x and $\tilde{x}$. The value of $|E|$ pertains only to the significand and in that respect gives the number of significant digits of accuracy for the approximation. From (2.12), $|E| \le 2^{-(m+1)}$ under rounding to the nearest value and $|E| \le 2^{-m}$ in the case of chopping.

As an example, let us revisit the approximation for 1/5 that was obtained using the simple eight-bit floating-point scheme of the previous section. In that case we had initially replaced 1/5 by .19921875 where

$$.19921875 = 1.10011 \times 2^{-3}.$$

This led to the representation of 1/5 as 01011010. The first sign bit is set to 0 and the two's complement representation of -3 (i.e., 101) occupies the next three bits. The final four bits, 1010, are used to approximate 10011. Chopping would replace this by 1001 while round-to-nearest value is what gave us 1010. The result is that chopping approximates .2 by .1953125 with an error of .0046875 ($< .0078125 = 2^{-7}$) while rounding to the nearest value approximates .2 by .203125 with an absolute error of .0031025 ($< .00390625 = 2^{-8}$). The relative absolute error is $.003125/.2 = .015625$ ($< .03125 = 2^{-5}$) for rounding to the nearest value and $.0046875/.2 = .0234375$ ($< .0625 = 2^{-4}$) for chopping.

We now wish to consider how the effect of rounding will propagate when performing the basic arithmetic operations of multiplication, division, addition and subtraction. Processors (at least those that comply with IEEE standards) use algorithms that perform arithmetic operations in ways that produce *exactly rounded* answers in a sense that will now be described. Details concerning these types of algorithms can be found in Koren (2002) and Lu (2004).

Let $\tilde{}$ denote the result of floating-point approximation so that, for example, $\tilde{x}$ is the floating-point approximation to a number x. Similarly, $\tilde{+}, \tilde{-}, \tilde{\times}$ and $\tilde{/}$ will be used to denote the floating-point implementation of the arithmetic operators $+, -, \times$ and $/$, respectively. Then, if $\tilde{x}$ and $\tilde{y}$ are floating-point representations of x and y, the operators $\tilde{+}, \tilde{-}, \tilde{\times}$ and $\tilde{/}$ produce exactly rounded results if

$$\begin{aligned} \tilde{x}\tilde{+}\tilde{y} &= \widetilde{\tilde{x}+\tilde{y}} \\ \tilde{x}\tilde{-}\tilde{y} &= \widetilde{\tilde{x}-\tilde{y}} \\ \tilde{x}\tilde{\times}\tilde{y} &= \widetilde{\tilde{x}\times\tilde{y}} \\ \tilde{x}\tilde{/}\tilde{y} &= \widetilde{\tilde{x}/\tilde{y}}. \end{aligned}$$

In words these relations mean that the operations on $\tilde{x}$ and $\tilde{y}$ produce the same answer as if the nonfloating-point operators $+, -, \times, /$ had been applied to the number $\tilde{x}$ and $\tilde{y}$ and the outcome was then rounded to obtain the final m-digit binary representation. If the operations are now assumed to produce exact rounding to the nearest value this leads to relations such as

$$\tilde{x}\tilde{+}\tilde{y} = (\tilde{x}+\tilde{y})(1+E) \tag{2.15}$$

and

$$\tilde{x}\tilde{\times}\tilde{y} = (\tilde{x}\times\tilde{y})(1+E) \tag{2.16}$$

for a generic relative error E satisfying $|E| \le 2^{-(m+1)}$.

Let us use (2.16) to bound the error incurred in computing the product $P_n = \prod_{i=1}^n x_i$ of real numbers $x_1, \ldots, x_n$. In this regard, there are two quantities to consider: the product $\tilde{P}_n = \prod_{i=1}^n \tilde{x}_i$ of floating-point approximations to the x_i and the approximation to $\tilde{P}_n$ (and P_n) provided by $\hat{P}_n = \tilde{x}_1 \tilde{\times} \cdots \tilde{\times} \tilde{x}_n$. An expression that relates $\hat{P}_n$ to $\tilde{P}_n$ can be obtained from the recursion that begins with $\hat{P}_1 = \tilde{x}_1$ and has the general step

$$\hat{P}_j = \tilde{x}_j \tilde{\times} \hat{P}_{j-1}$$

for $j = 2, \ldots, n$. From (2.16) one may conclude that

$$\hat{P}_n = \tilde{P}_n \prod_{j=2}^n (1+E_i)$$

for $|E_i| \le 2^{-(m+1)}, i = 2, \ldots, n$. By now writing

$$\prod_{j=2}^n (1+E_i) = 1+E$$

bounding the relative error $|\hat{P}_n - \tilde{P}_n|/|\tilde{P}_n|$ becomes tantamount to placing a bound on $|E|$.

Following Sterbenz (1974, Section 3.5) we have

$$-1 + \left(1 - 2^{-(m+1)}\right)^{n-1} \le E \le -1 + \left(1 + 2^{-(m+1)}\right)^{n-1}$$

with

$$\begin{aligned} -1 + \left(1 - 2^{-(m+1)}\right)^{n-1} &= \sum_{j=1}^{n-1} \binom{n-1}{j} (-2)^{-j(m+1)} \\ &\ge -\sum_{j=1}^{n-1} \binom{n-1}{j} 2^{-j(m+1)} \\ &= -\left\{-1 + \left(1 + 2^{-(m+1)}\right)^{n-1}\right\}. \end{aligned}$$

Thus,

$$\begin{aligned}|E| &\le -1+\left(1+2^{-(m+1)}\right)^{n-1}\\ &= -1+\sum_{j=0}^{n-1}\binom{n-1}{j}2^{-j(m+1)}\\ &\le -1+\sum_{j=0}^{\infty}\frac{(n2^{-(m+1)})^j}{j!}\\ &= -1+\exp\{n2^{-(m+1)}\}.\end{aligned}$$

So, if n is small compared to 2^{m+1}, the relative error in approximating $\tilde{P}_n$ by $\hat{P}_n$ will be similarly small and on the order of $n2^{-(m+1)}$.

A similar analysis can be applied using (2.14) to see that

$$\tilde{P}_n = P_n(1+\bar{E})$$

with $|\tilde{E}| \le -1+\exp\{n2^{-(m+1)}\}$. Putting both approximations together produces

$$\hat{P}_n = P_n(1+\bar{E}+E+\bar{E}E)$$

giving a relative absolute error bound on the order of $-1+\exp\{n2^{-(m+1)}\}$ for approximation of P_n. To the first order this bound would behave like $n2^{-(m+1)}$. Taking $n=2^k$ suggests a loss of one (binary) decimal of accuracy for about every two multiplications in a worst-case scenario.

The errors incurred from floating-point division can be bounded similarly to the case of multiplication (Exercise 2.5). However, the same is not true for floating-point addition and subtraction. In fact, as noted by Wilkinson (1963, page 17), it is not possible to obtain a bound for the relative errors associated with such calculations as the floating-point sum/difference can be zero when the true sum is not.

An indication of how problems can arise from subtraction is provided by taking x and y to be floating-point numbers with m digit significands having $x = 1.0\cdots02^p$ and $y = b_0.b_1\cdots b_m2^{p-1}$ with $b_0=\cdots=b_m=1$. In this instance the difference between x and y is $2^{p-(m+1)}$. This difference is approximated by 2^{p-m} for chopping and 0 under rounding to the closest value. The absolute relative error for approximating $x-y$ is one in either case because $(2^{p-(m+1)}-2^{p-m})/2^{p-(m+1)}=1$ and $(0-2^{p-(m+1)})/2^{p-(m+1)}=-1$. This implies that the absolute error in subtracting x from y using floating-point calculations can be as large in magnitude as the target quantity $x-y$. Put another way, there may be no digits of accuracy in the output unless steps are taken to provide additional accuracy in the calculation.

Suppose that the floating-point addition operator produces exactly rounded results and again let $\tilde{x}_1,\ldots,\tilde{x}_n$ be floating-point approximations of numbers $x_1,\ldots,x_n$. To compute an approximation $\tilde{S}_n$ to $\mathrm{S}_n=\sum_{j=1}^n\tilde{x}_j$ use the recursion that starts with $\tilde{S}_1=\tilde{x}_1$ and has the general step

$$\tilde{S}_j = \tilde{x}_j\tilde{+}\tilde{S}_{j-1}$$

for $j=2,\ldots,n$. At the jth step of the recursion

$$\tilde{S}_j = (\tilde{x}_j+\tilde{S}_{j-1})\,(1+E_j)$$

with $|E_j| \leq 2^{-(m+1)}$ under rounding to the closest value. This leads to

$$\begin{aligned}\tilde{S}_n &= \tilde{x}_n(1+E_n) + \tilde{S}_{n-1}(1+E_n)\\ &= \tilde{x}_n(1+E_n) + \tilde{x}_{n-1}(1+E_{n-1})(1+E_n) + \tilde{S}_{n-2}(1+E_{n-1})(1+E_n)\\ &= \sum_{j=1}^{n} \tilde{x}_j(1+\bar{E}_j),\end{aligned}$$

where $\bar{E}_n = E_n$,

$$1+\bar{E}_j = \prod_{k=j}^{n}(1+E_k)$$

for $j = 2, \ldots, n-1$ and $\bar{E}_1 := \bar{E}_2$. (The notation := that appears here and elsewhere throughout the text indicates that the expression on the right hand of := replaces or overwrites the one on the left.) Hence, $S_n - \tilde{S}_n = \sum_{j=1}^{n} \tilde{x}_j \bar{E}_j$ and, as before

$$|\bar{E}_j| \leq -1 + \exp\{(n-j)2^{-(m+1)}\}.$$

But, unlike multiplication, the values of the $\tilde{x}_j$ are inextricably linked into the relative approximation errors with the consequence that the bound depends on the order of summation. This suggests that larger errors will accumulate as multiples of the earlier terms that are entered into the summation. From this perspective the best strategy would seem to require that addition should proceed with values being summed in inverse order of their magnitude to minimize the effect of size disparities on the summation process.

To see an example of the effect of order on addition, consider summing the series $1/j^2$ from $j = 1$ to n for some integer n. The code below represents an attempt to carry out this calculation numerically.

Listing 2.3 *series.cpp*

```
//series.cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char* argv[]){
  float sumL = 0., sumU = 0.;
  int n = atoi(argv[1]);
  for(int j = 1; j <= n; j++){
    sumL = sumL + 1/(float)(j*j);
    sumU = sumU + 1/(float)((n + 1 - j)*(n + 1 - j));
  }
  cout << setprecision(8) << "Direct and reverse sums are "
       << sumL << " and " << sumU << endl;
  return 0;
}
```

The first new thing we see in this listing is the inclusion of the header files cstdlib and iomanip that, respectively, allow us to access some useful functions from C and output manipulators for formatting the printed output. In this instance iomanip furnishes the `setprecision` function that is used with argument 8 to have eight-decimal numbers written to standard output. This code also contains the first appearance of `argc` (for argument count) and `argv` (for argument vector) in the arguments to the `main` function. These are quantities that correspond to command line input. In particular, one can think of `char* argv[]` as being an array of character strings (actually, a pointer to memory that holds arrays of `char` variables) with each string being a non-white space component of the white-space delimited text that was entered on the shell command line. The first array element `argv[0]` contains the (./ prefaced) name of the executable while `argc` is the number of strings held in `argv`. Another new twist is the use of the function `atoi` that is made available with the cstdlib header. This function is used to transform a string of character values into an integer variable. The argument to `atoi` in this case is assumed to represent a sequence of digits that may be preceded by a sign. If the string cannot be interpreted as a number, `atoi` returns 0. This is a method of dealing with the fact that command line input necessarily comes in as character strings but will often be used to represent numeric (in this case integer) values. The function `atof` performs the same type of operation except that the transformation is to a floating-point representation.

To compute the sums in Listing 2.3 we need to calculate reciprocals of the expression `j*j` for a variable `j` of type `int` with `*` indicating multiplication. Unless precautions are taken to the contrary, such ratios will be evaluated using integer arithmetic with the consequence that a value of 0 will be returned for `1/(j*j)` for every `j` that exceeds 1. The way to circumvent this behavior is through the use of another cast. Here this takes the form `(float)(j*j)` which computes the integer product and converts the outcome to type `float`.

Our program sums the series in direct and reverse order to produce the two sums: `sumL` and `sumU`. The two values will necessarily agree if they can be evaluated without error. This is not the case for finite precision arithmetic as indicated by the results from running the program (compiled with the name series).

```
$ ./series 5000
Direct and reverse sums are 1.6447253 and 1.644734
```

As noted above, for type `float` the best one might expect for accuracy is seven decimal digits. But, the two ways of computing the sum have differences that appear in the fifth decimal place when there are $n = 5000$ terms in the sum. To determine which answer is closer to the actual sum, the computations can be redone in double precision: i.e., every `float` designation in series.cpp is replaced with `double`. This altered set of code (compiled as dseries) produced the result

```
$ ./dseries 5000
Direct and reverse sums are 1.6447341 and 1.6447341
```

As expected, computing the sum in reverse order where smaller values are added in first provides the most accuracy. The approximate sums can be compared to $\sum_{j=1}^{\infty} j^{-2} = \pi^2/6 \doteq 1.644934$.

2.6 Computing a sample variance

One of the earliest calculations carried out in an elementary statistics methods course is computing a sample mean and standard deviation from a set of data values $x_1, \ldots, x_n$. The sample standard deviation is defined to be

$$S = \sqrt{\mathrm{RSS}/(n-1)},$$

where RSS is the residual sum-of-squares

$$\mathrm{RSS} = \sum_{j=1}^{n} (x_j - \bar{x})^2$$

with $\bar{x} = n^{-1} \sum_{j=1}^{n} x_j$ the sample mean.

A direct application of the definitional formula would suggest a two-pass algorithm where $\bar{x}$ is obtained first and then used to calculate the RSS. A pseudo-code description of the recursive process is given below.

Algorithm 2.1 Two-pass algorithm for the standard deviation

```
$\bar{x} = 0$
for $j = 1$ to $n$ do
    $\bar{x} := \bar{x} + x_j$
end for
$\bar{x} := \bar{x}/n$
$\mathrm{RSS} = 0$
for $j = 1$ to $n$ do
    $\mathrm{RSS} := \mathrm{RSS} + (x_j - \bar{x})^2$
end for
return $\sqrt{\frac{\mathrm{RSS}}{n-1}}, \bar{x}$
```

As an alternative to the two-pass formula, it is standard pedagogical practice to describe the use of a "computational" formula for RSS that stems from the identity

$$\mathrm{RSS} = \sum_{j=1}^{n} x_j^2 - n^{-1}\left(\sum_{j=1}^{n} x_j\right)^2 .$$

The resulting computations then proceed along the lines of our next algorithm.

Algorithm 2.2 "Computational" algorithm for the standard deviation

```
$T = \mathrm{RSS} = 0$
for $j = 1$ to $n$ do
    $T := T + x_j$
    $\mathrm{RSS} := \mathrm{RSS} + x_j^2$
end for
$\mathrm{RSS} := \mathrm{RSS} - T^2/n$
return $\sqrt{\frac{\mathrm{RSS}}{n-1}}, T/n$
```

This latter approach has the advantage of requiring only one pass through the data. The disadvantage is that $\sum_{i=1}^{n} x_i^2$ and $(\sum_{i=1}^{n} x_i)^2/n$ can agree across a number of significant digits. This produces inaccuracies when carrying out the subtraction at the end of the `for` loop for reasons that were discussed in the previous section.

Chan and Lewis (1979) define the condition number associated with S to be

$$\kappa = \sqrt{1 + n\bar{x}^2/\mathrm{RSS}}.$$

If S is small relative to $|\bar{x}|$, κ is approximately $\bar{x}/S$. This latter quantity is recognized as the inverse of the sample coefficient of variation that provides a scale-free measure of data variation. In this setting small values for the coefficient of variation can be seen as indicative of instances where the "computational formula" may be problematic. In such cases the data will consist of values that are all (relatively) close to their mean value with the consequence that $\sum_{j=1}^{n} x_j^2$ and $n^{-1}\left(\sum_{j=1}^{n} x_j\right)^2$ may be sufficiently similar to create cancellation problems.

Suppose now that the original data $x_1, \ldots, x_n$ are replaced by $x_1(1+E_1), \ldots, x_n(1+E_n)$ as might occur on a computer through the use of floating-point approximations. Then, Chan and Lewis (1979) establish that the standard deviation $\tilde{S}$ computed (without round-off error) from the altered data will satisfy $\tilde{S} = S(1+\delta)$ with

$$|\delta| \leq \kappa\gamma + O\left(n\left(\kappa\gamma\right)^2\right) \tag{2.17}$$

for

$$\gamma = \max_{1\leq j\leq n} |E_j|.$$

For small γ the lead term in the bound on $|\delta|$ is linear in κ which indicates that an increase in the round-off error in the data will be met by a proportional increase in the (bound for the) relative absolute error for computing S with the condition number representing the proportionality factor. We can then view κ as a measure of the inherent sensitivity of S for a particular data set to the effect of floating-point approximation error.

The analysis that led to (2.17) was in the ideal setting where the computation of $\tilde{S}$ can be carried out without error. Of course, the ideal case does not hold in practice and additional errors will be introduced through whatever algorithm is used to compute an approximation to $\tilde{S}$. To measure this let $\tilde{S}_c$ be the approximation to S that is obtained by applying a computational algorithm to $x_1(1+E_1), \ldots, x_n(1+E_n)$. We then define the relative absolute error to be

$$\mathrm{RAE} = |\tilde{S}_c - S|/S. \tag{2.18}$$

This quantity will be used to compare the accuracy of the two-pass and "computational" algorithms for computing a sample standard deviation that were discussed above. The condition number κ will play a central role in these comparisons.

Chan and Lewis (1979) give an approximate (ignoring higher-order terms) upper bound of the form

$$\mathrm{RAE} \leq 2\kappa\gamma + \left(\frac{n}{2}+1\right)\gamma$$

for the two-pass algorithm. A similar bound for the "computational formula" is

$$\mathrm{RAE} \leq \gamma + \left(\frac{3}{2}n+1\right)\kappa^2\gamma.$$

Figures 2.1–2.2 give plots of the (base 10 logarithm of the) bounds for the two-pass and "computational" formula, respectively. In doing this we took $n = 100$, $\gamma = 2^{-24}$ (as would be expected for single precision round-off errors), chose the true mean for the data to be one and then let the true standard deviation range from 10^{-5} to 1. For simplicity κ is approximated here by σ^{-1} and the horizontal axis is in terms of $-\log_{10}\sigma$ in the figures.

The suggestion from Figures 2.1–2.2 is that the "computational formula" will be less accurate for data sets with small coefficients of variation or large values of κ. This is a consequence of the fact that the "computational" formula's bound involves κ^2. In fact, the difference between the bounds for the "computational" and two-pass algorithms is

$$\frac{1}{2}n\gamma[\kappa^2-1] + \kappa\gamma[(n+1)\kappa - 2].$$

This is always nonnegative because $\kappa \geq 1$. Of course, the fact that only upper bounds are involved means that comparisons of this nature are not conclusive.

A small empirical study was conducted to further investigate the relevance of the upper bound comparisons for use of the two computing algorithms. Data were generated from normal distributions with means of one and standard deviations σ for which $\log_{10}(\sigma) = (j-10)/2, j = 0, \ldots, 10$. To accomplish this the Wichmann-Hill algorithm from Section 4.5 was used to generate uniform random deviates that were then transformed to normality using the Box-Muller transformation treated in Section 4.7. For each value of $\kappa = 1/\sigma$, 100 replicate samples of size $n = 100$ were generated and their standard deviations were

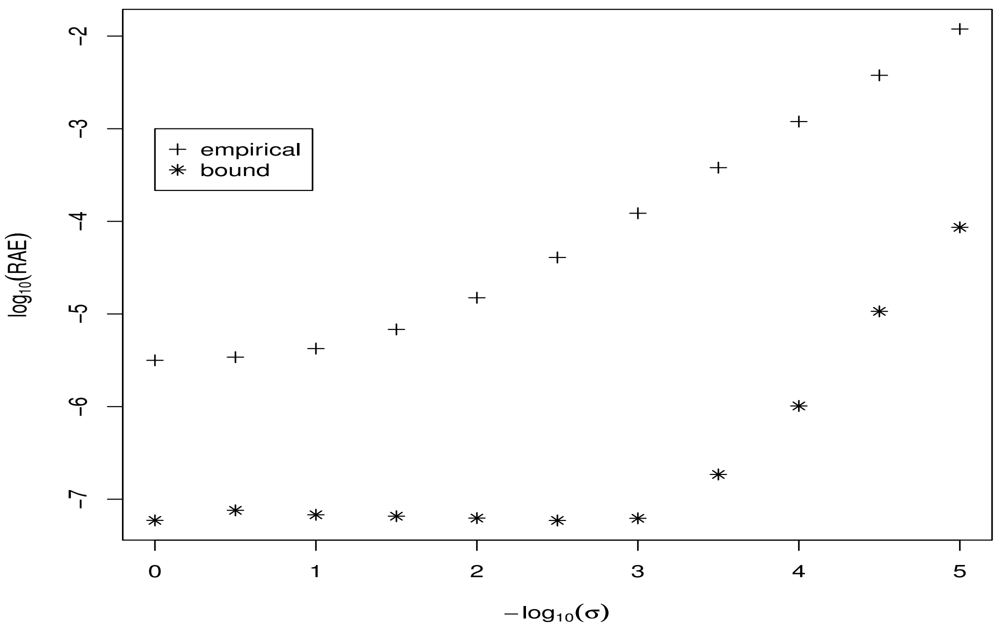

Figure 2.1 *Bounds and empirical results for the two-pass algorithm*

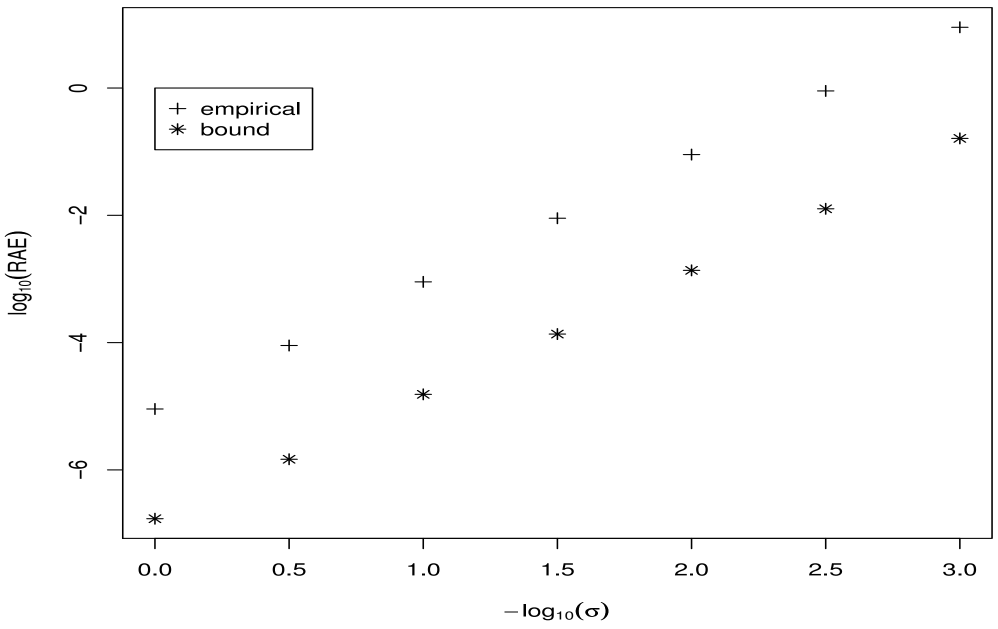

Figure 2.2 *Bounds and empirical results for the "computational" algorithm*

computed via the two-pass and “computational” algorithms. The calculations were carried out in single precision and then compared to a “true” value that was computed in double precision using the two-pass approach. This gives 100 values for RAE for each of the two methods for computing S which were then averaged to produce the results depicted in the plots. From this we can see that the bounds are quite conservative but are accurate in predicting the general form of the relationships between RAE and κ. In the case of the “computational” algorithm, only those RAEs for values of κ between 1 and 10^3 are reported; negative values for RSS began to arise for cases with $\kappa > 10^3$. The basic conclusions derived from the upper bounds and empirical work therefore seems to coincide with our original intuition in suggesting that the two-pass algorithm is more accurate and should be preferred for computations.

There are also updating algorithms that can be used to compute RSS. For example, an algorithm developed by West (1979) and others (see, e.g., the discussion in Chan, et al. 1983) takes the form

Algorithm 2.3 West algorithm for the standard deviation

```
$\bar{x} = x_1$
RSS $= 0$
**for** $j = 2$ to $n$ **do**
  RSS $:=$ RSS $+ \frac{j-1}{j}(x_j - \bar{x})^2$
  $\bar{x} := \bar{x} + (x_j - \bar{x})/j$
**end for**
**return** $\sqrt{\frac{\text{RSS}}{n-1}}, \bar{x}$
```

The Chan/Lewis upper bound for this method is

$$RAE \leq \left(\frac{n}{2} + 2\right)\gamma + \left(\frac{\sqrt{2}}{3}n + 7\sqrt{n} + 1\right)\kappa\gamma.$$

Like the two-pass method, the error bound for West’s algorithm involves only κ rather than κ^2. The analog of Figures 2.1–2.2 that applies to the West algorithm is shown in Figure 2.3. The empirical results shown in the figure were obtained using the same simulation methods that were employed to produce Figures 2.1–2.2. The two-pass method is clearly less sensitive to growth in κ although both it and the West algorithm appear to behave similarly when κ becomes large.

2.7 Storage in R

To conclude this chapter let us mention a few things about the data types and precision of computations in R. First, the “primitive” data types in R include character, double, integer and logical. The double and integer types are analogs of the types with the same names in C++ while logical corresponds to the C++ `bool` data type. The character designation in R indicates a variable that holds character strings rather than just a single character as in C++. The double type in R also goes by the equivalent, and more frequently used, name of *numeric*. The most basic data structure in R is an array comprised of one of the primitive data types that is referred to as an *atomic vector*.

To access the storage mode of a given object, one uses either the `mode` or `storage.mode` functions. In terms of storage of numeric, non-integer values, R purports no single precision data type and all real numbers are stored in a double precision (eight-byte) format. Machine specific details concerning storage, etc., are held in the R list variable `.Machine`. For

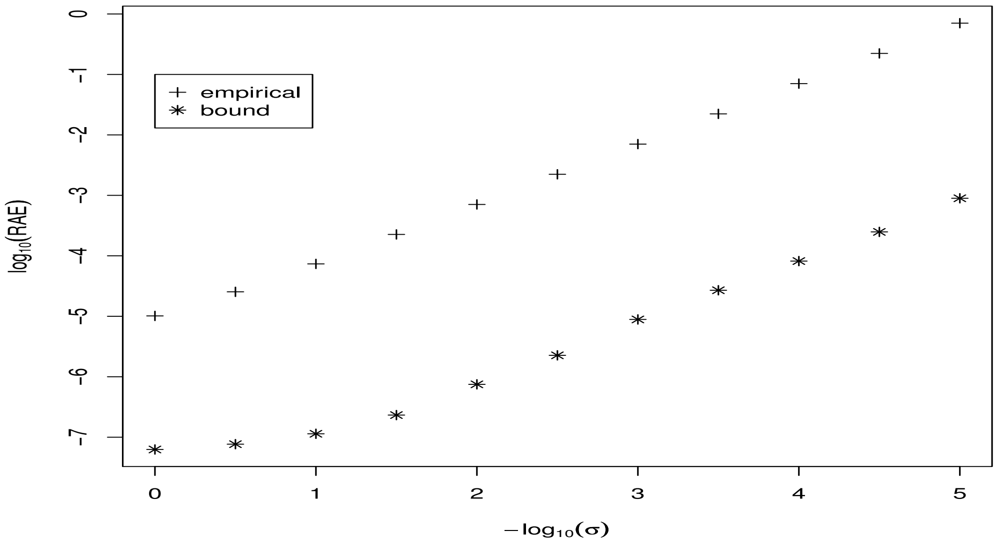

Figure 2.3 *Bounds and empirical results for the West algorithm*

example, for the machine that was used to create portions of the material in this chapter the `.Machine` information appears as

```
> noquote((format(.Machine)))
           double.eps          double.neg.eps            double.xmin
         2.220446e-16            1.110223e-16           2.225074e-308
          double.xmax             double.base          double.digits
        1.797693e+308                       2                      53
      double.rounding            double.guard      double.ulp.digits
                    5                       0                     -52
double.neg.ulp.digits         double.exponent         double.min.exp
                  -53                      11                   -1022
       double.max.exp             integer.max            sizeof.long
                 1024              2147483647                       4
      sizeof.longlong       sizeof.longdouble         sizeof.pointer
                    8                      16                       4
```

Some of the names for the list components are familiar from our discussions in Section 2.2 while the meanings of others can be deciphered by looking at the R help page for `.Machine`. In particular, the value of 53 for the `double.digits` variable indicates that the significand for a `double` is composed of 53 bits (i.e., 52 bits plus an implied unit bit for a normalized significand) which corresponds to the IEEE floating-point standard.

There were a couple of useful features of the R language that were used to produce the printed representation of the information in `.Machine`. First, the `format` function arranges its input for "pretty printing". The `noquote` function then suppresses the use of quotes in the printed output.

2.8 Exercises

2.1. Show that the largest integer that can be stored in m bits is $2^m - 1$.

2.2. Write, compile and run C++ code that will write `"Hello World!"` to the shell. Then, alter your program to where it will greet the world using your first name. To accomplish this use `argv` as in Listing 2.3. If you enter your first name on the command line, it will be available in `argv[1]` so that you can string `"Hello World from"` and `argv[1]` together using the output insertion operator `<<`. How would you modify this to use your whole name in the greeting?

2.3. Let x be a number in [0, 1) and let $\lfloor y \rfloor$ denote the greatest integer less than or equal to the real number y.

a) Show that the following algorithm will return the first m terms $b_1, \ldots, b_m$ in the binary representation for x. (Conte and de Boor 1972)

Algorithm 2.4 Binary representation algorithm

```
$c = x$
for $j = 2$ to $m$ do
  $b_j = \lfloor 2c \rfloor$
  $c := 2c - b_j$
end for
```

b) The C++ function for $\lfloor\rfloor$ is `floor`. Use this to create a program that will take the value of a `float` variable `x` with a value in [0, 1) as input and return the first five coefficients in its binary expansion (2.2). Note, to use `floor` you will need to include the math library with the statement `#include <cmath>`.

2.4. Another bit-wise operator is the exclusive OR or XOR operator `^`. If `k1` and `k2` are two integers, the bits for `k1^k2` are 1 whenever one or the other of the corresponding bits in `k1` or `k2` is 1 and are 0 otherwise.

a) Develop a binary addition algorithm based on `XOR`. [Hint: Incorporate a carry bit into the addition of two bits.]

b) Implement your algorithm from part a) in C++. Assume that you are dealing with numbers that are one byte in length and carry out your calculations using `unsigned short` integers.

2.5. Consider the division of two sets of floating-point numbers $\tilde{x}_1, \ldots, \tilde{x}_n$ and $\tilde{y}_1, \ldots, \tilde{y}_k$ with m-bit significands.

a) Assuming exact rounding, show that

$$(\tilde{x}_1 \cdots \tilde{x}_n)\tilde{/}(\tilde{y}_1 \cdots \tilde{y}_k) = [(\tilde{x}_1 \cdots \tilde{x}_n)/(\tilde{y}_1 \cdots \tilde{y}_k)](1+E)$$

with

$$\left(1 - 2^{-(m+1)}\right)^{(k+n-1)} \le 1 + E \le \left(1 + 2^{-(m+1)}\right)^{(k+n-1)}.$$

b) Use the bound for $1+E$ obtained in part a) to show that the relative error from division should behave like a constant multiple of $(n+k)2^{-(m+1)}$ if $n+k$ is small relative to 2^{m+1}.

2.6. Let $\tilde{}$ denote the result of floating-point conversion. Prove or disprove that $(\tilde{x}\tilde{+}\tilde{y})\tilde{+}\tilde{z} = \tilde{x}\tilde{+}(\tilde{y}\tilde{+}\tilde{z})$.

2.7. Let $y = 1 + x$ for a positive number x. If y is stored as the `float` value $\tilde{y}$ according to the IEEE standard, under what conditions will $\tilde{y} = 1$?

2.8. Under the IEEE standard the significand for a `double` is allocated 52 bits of storage.

a) Give bounds for the corresponding relative error associated with rounding to the nearest value and chopping for doubles.

b) How many decimal places of accuracy can be expected for a `double` under this storage scheme?

c) Can you conclude from this storage specification (why or why not) that summing the series $1/j^2$ in reverse order as in Section 2.5 except with double precision is accurate to seven decimals?

2.9. Show that upon completion Algorithm 2.3 will return the sample residual sum-of-squares.

2.10. Show that the algorithm below, due to Youngs and Cramer (1971), computes the sample residual sum-of-squares.

Algorithm 2.5 Youngs and Cramer algorithm

```
$T = x_1$
RSS $= 0$
**for** $j = 2$ to $n$ **do**
  $T := T + x_j$
  RSS $:=$ RSS $+ \frac{(jx_j - T)^2}{j(j-1)}$
**end for**
**return** $\sqrt{\frac{\text{RSS}}{n-1}}, \frac{T}{n}$
```

2.11. Consider the quadratic polynomial $q(x) = x^2 - 111.11x + 1.2121$.

a) Find the roots of q using the quadratic formula but using only five decimal digits with chopping in all your calculations (addition, multiplication, square root, etc.).

b) For a quadratic polynomial ax^2+bx+c show that the roots x_1 and x_2 satisfy $x_1x_2 = c/|a|$. Use this latter relationship to evaluate the root having smaller absolute value for q under the same conditions for digit retention and rounding and compare the result with your previous answer.

(Conte and de Boor 1972)