2. Data

In this section we will consider some of the data types supported by most computers along with the operations that are commonly performed on them. At the base are bytes and words — unstructured objects of fixed length. The operations that can be performed with them are bitwise logical operations and shift operations. These raw words can be regarded as containers that hold more structured types, of which the most important are characters, integers, and floating-point numbers. We will begin our discussion with words. We conclude with a discussion of operation counts as a means of comparing algorithms.

2.1. Bits, bytes, and words

Synopsis: A byte consists of eight bits. Words consist of a number of bytes. Nomenclature differs, but typically a word consists of four bytes (16 bits), a half word of two bytes, and a double word of eight bytes.

There are two conventions for how the bytes in a word are stored in memory. Big endian machines store them in order beginning with the most significant byte; little endian store them in the reverse order beginning with the least significant byte. The endian type usually, but not always, makes no difference in the way you code.

Words in memory are generally aligned so that the nest within each other. Thus a four byte word will have an address in which its trailing two bits are zero. Alignment makes for efficient retrieval of words, but it can cause memory fragmentation.

Most machines provide two classes of operations on words: logical and shift. Typical logical operations are 'or', 'and', and 'xor' (exclusive or). The shift operations shift the bits of a word to the left or right. Bits that fall off one end are lost and the bits at the other end are set to zero. These two classes of operations can be used to isolate fields in a word.

—

2.1.1. Words

As we have seen, a **bit** is a single binary digit that can take on one of the two values 0 and 1. A **byte** is a group of eight bits. Since a hexadecimal digit can be represented by four bits, it is natural to describe bytes by pairs of hexadecimal digits. For example, the byte 01011110_2 may be represented by the number $5E_{16}$, the 5 corresponding to the half byte 0101 and the E corresponding to 1110. The subscripts 2 and 16 refer to the base of the representation — binary and hexadecimal respectively. (When the base is clear, we will drop the subscripts).

Words consists of a sequence of bytes. The following table gives typical word sizes in bytes and bits.

name	bits	bytes
byte	8	1
short or half word	16	2
word	32	4
long or double word	64	8

The nomenclature for words is complicated and varies from system to system. With the advent of machines with standard 64 bit registers one might expect a 'word' to consist of eight bytes and a double word of sixteen. But a likely possibility is that a 'word' will continue to have four bytes, and the 16 byte word will be called a quad word.

A further complication is that most programming languages used in scientific computing do not recognize words as such. Instead they have integer types of various sizes. To define a variable representing a word in such a language you must declare it to be an integer, even though you have no intention of performing arithmetic operations with it.

2.1.2. Words in memory

The way words are stored in memory also varies with the machine. Most computers address memory by bytes, as is shown in Figure 1.2. On such machines the representation of a byte in memory is unambiguous. On the other hand, words consist of several contiguous bytes. This raises two questions. First, where are the word boundaries; i.e., where to words start in memory? Second what is the order of the bytes within a word? We will consider each of these questions in turn.

In principle, we could in principle begin storing a word at any address. But it is more efficient to store words so that they always nest within one another — i.e., so that a double word consists of two single words, which in turn consists of two half words. This can be done by requiring that the address of the words be in **alignment.** The following table, in which we assume a four byte word, shows the low order bits of the addresses of properly aligned objects.

object	address
byte	`xx...xxxx`
half word	`xx...xxx0`
word	`xx...xx00`
double word	`xx...x000`

(2.1)

A byte can have an arbitrary address. The address of a half word is a multiple of two; the address of a word is a multiple of four; and the address of a double word is a multiple of eight.

To appreciate the efficiencies of alignment, we must understand that although memory is addressable by bytes, what is usually transferred between the CPU and memory

are words — call them memory words. Moreover, memory words are aligned. For example, if memory words are four bytes long, then their addresses are multiples of four. Since any half word or byte aligned according to (2.1) lies within a memory word, it can be transferred with one memory reference. On the other hand, a half word with an address of the form `xx...x11` has one byte in a memory word its other byte in the next memory word. Hence it would require two memory transactions to transfer the half word.

The efficiencies of alignment do not come without a price. If we have a sequence A, B, C consisting of a four-byte word, a half word, and another four-byte word word, then they must be stored in main memory as follows.

low order bits of the address	memory
x...x00	A
x...x10	
x...x00	B
x...x10	X
x...x00	C
⋮	⋮

(2.2)

From this we see that the two bytes, labeled X, between B and C are wasted. This is an example of **memory fragmentation**, to which we will return later. To anticipate a little, arrays of words of the same length pose no fragmentation problems, but arrays of ill-planned C structures or Fortran derived types can result in large amounts of unused, unavailable memory.

Turning now to the ordering of bytes in a word, we first note that we could store its bytes in any order, as long as the particular ordering is consistent from word to word. In practice, there are only two orders: big endian and little endian.[2]

Consider a four byte word ABCD. The byte A is called the **leading** or **most significant** byte. The byte D is called the **trailing** or **least significant** byte. In **big-endian** representation the bytes are stored in increasing memory locations beginning with the leading byte. In **little-endian** representation the bytes are stored in increasing memory locations beginning with the trailing byte. The following table illustrates the two forms of storage beginning at the address `a`.

[2]These terms are from *Gulliver's Travels* by Jonathan Swift, who tells of a war between Lilliput and Blefuscu. The nominal cause was a disagreement about how to open an egg. The Lilliputians favored cracking it at the little end; the Blefuscudians at the big end. The big endians and little endians of today, though not actually at war, live in a state of uneasy accommodation.

address	BE	LE
a	A	D
a+1	B	C
a+2	C	B
a+3	D	A

(BE = Big Endian; LE = Little Endian)

Both conventions are widely used. For example, Intel machines are little endian while SUN machines are big endian. MIPS machines allow you to choose your endian type. In most scientific programs the difference is invisible to the coder. The reason is that the endian distinction is purely a matter of how things are stored in memory. When words are loaded into the machine they end up in registers in their natural order. For example, consider the word

$$47A5_{16} = 0100011110100101_2$$

on a machine with words that are two bytes in length. In big endian order it will be stored consecutively in memory as 47 and A5. In little endian the order will be A5 and 47. But when they are loaded into a register, the machine adjusts for its endian type and the register contains 0100011110100101 for both.

There are two important exceptions to this invisibility. First, some programming languages — notably C — allow you to address the individual bytes in a word as they are stored in memory. Obviously, the order in which the bytes are stored will make a difference in what you see. Second, data transfer between machines generally takes place in the order of the data in the memory of the transferring machine. If two machines differ in their endian types, bytes from one will arrive in the wrong order for the other. This problem is especially important in networks of heterogeneous computers such as the Internet.

2.1.3. Operations on words

There are few meaningful operations on words. The reason is that a raw word is simply a sequence of bits with no structure to give an operation a grip. It is not surprising then that general instructions for manipulating words work at the bit level. Of these the most important are bitwise logical operations and shift operations.

The **bitwise logical operations** combine the corresponding bits of two words to give another word. The following table shows the three principle operations: **and**, **or**, and **xor (exclusive or)** for a four bit word (which is sufficiently long to illustrate all possible combinations).

```
  0011        0011        0011
or 0101   and 0101   xor 0101
  ----        ----        ----
  0111        0001        0110
```

A fourth operation, called complementation, flips the bits of of a word. However, complementation can be done by computing the xor of the word with a word of all ones.

The **logical shift operations** are best illustrated by an example. The number 10110110 shifted right by three bits is 00010110. Shifted left by three bits it becomes 10110000. The bits that are shifted in, are always zero. The bits that are shifted out are lost. Thus, a right and left shifting 10110110 by three bits gives 10110000. (Exercises ???? explore an arithmetic right shift that behaves differently.)

A combination of bitwise operations and shifts can be used to isolate fields in a word. For example, we can isolate the `B` in the two byte word `a = x35B8` by the following steps. (The `x` in `x35B8` indicates that the number is hexadecimal.)

```
1. b = And(a, x00F0)
2. b = ShiftRight(b, 4)
```
(2.3)

The first operation is called **masking** and produces the word `x00B0`. The subsequent shift moves the B to the right end of the word to give `x000B`.

All machines have some complement of logical and shift instructions. Programming languages differ in how they provide access to these instructions. In C the operations are primitive operations of the language itself. For example, the statement

```
b = (a & 0x00F0) >> 4;
```

has the same effect as (2.3). Fortran 95 accomplishes the same thing with functions.

```
b = shift(iand(a, mask), -4)
```

where `mask` contains `z'00F0` (`z'` being the prefix indicating a hex constant).

2.2. CHARACTERS

SYNOPSIS: For a computer to manipulate characters, such as 'a', ')', or ' e', they must be represented in binary. A character code is a convention that associates a unique sequence of bits with characters. An widely used code is the extended ASCII code, in which each of 256 characters is represented by one of the 256 possible bytes. There are many different (and incompatible) character codes, and for this reason machines do not generally provide instructions to manipulate characters, leaving the job to software.

A sequence of characters is called a string. The representation of strings generally depends on the high-level programming programming language in which one is coding.

—

In addition to crunching numbers, computers must work with letters, digits, and other characters. The CPU ultimately works only with bits, and hence characters must be coded as sequences of bits. A good example of a **character code** is the **extended ASCII** (American Standard Code for Information Interchange) character code, in which

one byte represents one character. For example, the byte whose hexadecimal value is 41_{16} represents the letter A. Again, the byte whose value is 89_{16} represents the letter 'e' with a dieresis — ë.

The ASCII character set is not confined to representing letter and numbers. For example, the following array shows a sequence of bytes (in hexadecimal) with their interpretations over them.

```
 y    =    x  /  (  1  -  x  )
79 20 3D 20 78 2F 28 21 2D 79 29
```
(2.4)

Note that there is a special code 20_{16} for a space. There are also codes for control characters. For example, $0A_{16}$ represents a line feed — the signal to begin a new line.

In principle, codes like the extended ASCII code are arbitrary. In practice, a lot of thought goes into devising a code. In is no coincidence, for example, that letters a–z are numbered consecutively (viz., 61-7A). It makes alphabetizing a set of words equivalent to sorting the numerical values of their bytes.

There are many character codes. The original ASCII (aka 7-bit ASCII) refers to the first 128 characters in the extended ASCII code, specifically characters numbered x00–x7F. This set has been extended in different ways. A particularly important extension is the ISO8859-1 (Latin-1) code, which provides the characters to print western European languages and is the basic HTML character set. Unicode is an ambitious attempt to represent the characters of all languages (including such languages as Chinese) in a unified code.

Because of the variety of character codes, CPU's do not usually have special instructions for dealing with them. Instead they perform sequences of shifts and masks to isolate characters and comparisons to determine particular characters. Most programming languages provide functions implementing these manipulations. For example the C function `isdigit(c)` tests if the integer `c` represents a decimal digit.

A sequence of characters, like that in (2.4), is called a **string.** Just like characters, strings have different representations, which usually depend on the programming language.

2.3. INTEGERS

SYNOPSIS: On a machine nonnegative integers can be represented by regarding the bits in a word as a binary number, that is, an unsigned integer. Integers can be added, subtracted, multiplied, and divided. However, the result of these operations cannot always be represented in the computer. This situation is called an arithmetic exception. Arithmetic exceptions can be handled by an automatic default or by trapping to an exception handler. However, the latter may be computationally expensive.

Addition and subtraction are the fastest operations. Multiplication can be almost as fast as addition. Division is much slower. However, multiplication and division by two can be implemented

by shifting.

There are two ways of representing both positive and negative integers: signed-magnitude representation and 2's complement representation. Signed magnitude representation is the more natural, but it has two representations for zero. Unsigned magnitude is often preferred because it makes addition easier to implement. However, it is asymmetric in the sense that there are more negative integers than positive.

—

In scientific computing numbers come in two flavors: integers or fixed-point numbers and floating-point numbers. One usually thinks of number crunching in terms of floating-point number, but integer arithmetic, especially in the form of indexing, can account for a significant part of the overhead in a numerical algorithm.

2.3.1. Unsigned integers

Integers come in different sizes, usually corresponding to the sizes of the words supported by the machine in question. They also have various representations. The simplest is the **unsigned integer.** Here the bits in a word are taken as the bits of a binary number, which associates a value with the word. For example, the configuration of bits 01101001 of eight bits has the value

$$01101001_2 = 0{\cdot}2^7 + 1{\cdot}2^6 + 1{\cdot}2^5 + 0{\cdot}2^4 + 1{\cdot}2^3 + 0{\cdot}2^2 + 0{\cdot}2^1 + 1{\cdot}2^0 = 105_{10}.$$

Here the subscripts denote the bases of the representation.

2.3.2. Operations and exceptions

All CPU's provide the standard arithmetic operations of addition, subtraction, multiplication, and division. These operations are known to us all and at first glance appear unproblematical. However, they are all subject to **arithmetic exceptions** which can caused a program to fail. Let us look at this matter in more detail for the case of addition.

For definiteness, suppose we are operating with four bit unsigned integers. Addition presents no problems if the answer can be represented by four bits; however, if it cannot be so represented, we have a condition called **overflow.** For example, consider the sum of 13 and 5.

$$\begin{array}{rr} & 1101 \\ + & 0101 \\ \hline & 10010 \end{array}$$

The problem is that this result cannot be stored as a four bit unsigned integer. The question is then what to do. The answer is that no one convention serves all purposes.

One possibility is to discard the extra bit. In our example this would give the answer $0010_2 = 2_{10}$ which is just $13 + 5 \pmod{16}$. In many applications this is just what we want.

Another possibility is to have the CPU **trap** on overflow. A trap is an internal interrupt that is initiated by the CPU. Just like an interrupt, the trap transfers control to a trap handler that takes appropriate action. For example, it may declare the overflow to be a **fatal error** and cause the operating system to terminate the offending process.

In some respects, trapping appears to be an ideal solution. The user furnishes a handler that decides whether the result is to be returned modulo 16 (in our particular example) or if the exception is a fatal error. But there are two difficulties with this approach.

The first is that some commonly used programming languages have no mechanism to allow the program to write a trap handler. Thus the decisions of what to do is left to the system, not the programmer. The second difficulty is that trapping and handling take time. If the addition in question occurs in an inner loop and overflow is frequent, the execution of the program could be slowed down significantly. A possible compromise it to provide the programmer with a way to turn off the trap and the compute result modulo 16. But this may favor some users — those who want to add modulo 16 — over others who want to do something else. These considerations show that exception handling is not a trivial undertaking.

Here is a list of possible exceptions for an operations with unsigned integers.

`a + b:`	Overflow
`a - b:`	Negative result, i.e., `a` $<$ `b`
`a*b:`	Overflow
`a/b:`	Division by zero or noninteger result

As with addition, there is no general agreement on how to handle the exceptions for subtraction and multiplication. For division it is generally agreed that division by zero is fatal (but not necessarily in floating-point arithmetic; see p 27). There is also agreement about what to do when the result is not and integer — as when we calculate $17/3$ — namely the exact quotient should be truncated toward zero. For example

$$17/3 = 5.6666666\ldots \rightarrow 5,$$

where the right arrow denotes the truncation. Note that this result corresponds to what we were taught to say in elementary school: Seventeen divided by three gives five with remainder two.

2.3.3. Timing

The times required to perform the four arithmetic operations vary. Typically, addition and subtraction are cheapest. The cost of multiplication can be brought near that of

addition. Division is the most expensive operation. Fortunately, in many calculations adds and multiplies predominate.

There is one case where division is cheap. Consider the two numbers

$$105_{10} = 01101001_2 \quad \text{and} \quad 105_{10}/4_{10} \to 25_{10} = 00011010_2.$$

Here we assume that the quotient is truncated as described above. The second binary number can be obtained from the first by performing a logical right shift of two bits. The natural generalization holds: a logical right shift of k bits is equivalent to division by 2^k. In general logical shifts are much cheaper than divisions. In the other direction, a left shift by k bits is equivalent to multiplication by 2^k. Thus shifts can be substituted for multiplication by powers of two.

2.3.4. Signed integers

Unsigned integers can only represent nonnegative numbers. To represent both nonnegative and negative numbers we must code the sign into the representation. The two most common ways of doing this are called signed magnitude representations and 2's complement representation.

Signed-magnitude representation reserves a bit, usually the most significant, to represent the sign — 0 denoting plus and 1 minus. The remaining bits of the word are taken to be an unsigned integer representing the magnitude of the number. For example, in an eight bit signed-magnitude representation the number 25_{10} is represented by 00001101 and -25_{10} by 10001101. This is a natural representation, but it has an unnatural consequence. The number zero has two representations: $+0 = 00000000$ and $-0 = 10000000$.

A widely used alternative is **2's complement** representation. Specifically, in an n bit word, the nonnegative numbers x ($0 \le x \le 2^{n-1}-1$) is its usual binary representation (with the leading bit zero). The negative number $-x$ ($0 < x \le 2^{n-1}$) is represented by the binary representation of $2^n - x$. Here is a table of four-bit 2's complement representations.

x	$+x$	$-x$
0	0000	
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8		1000

(2.5)

Two's complement representation seems unnatural, but in fact it is often preferred because it makes addition easier to implement in silicon. It shares with signed magnitude representation the fact that negative numbers begin with the bit 1. Zero is uniquely represented. But the representation is asymmetric in that -2^{n-1} is represented but 2^{n-1} is not.

Signed numbers can overflow or underflow. Because of the asymmetry of 2's complement representation, the largest negative number can overflow when it is multiplied or divided by -1.

2.4. IEEE FLOATING-POINT ARITHMETIC

SYNOPSIS: Floating-point arithmetic was introduced to compensate for the limited range of integer arithmetic. The prototypical floating-point number occurs in the scientific notation display of a hand-held calculator; e.g., -3.76542 -03. The value of this number is $-3.76542 \cdot 10^{-03} = -0.00376542$. In this representation, 10 is the base, -3.76542 is the significand and -03 is the exponent. The number is normalized so that its leading digit is nonzero.

IEEE binary floating-point arithmetic is today's standard. It consists of a single-precision (32-bit) representation and a double-precision (64-bit) representation. Single precision represents numbers ranging roughly from 10^{-38} to 10^{38} to an accuracy of about seven decimal digits. Double precision represents numbers ranging roughly from 10^{-307} to 10^{307} to an accuracy of about sixteen decimal digits.

The inaccuracy inherent in floating-point representation is called rounding error. The IEEE standard requires that all arithmetic operations return a correctly rounded answer. In extended computations, these inaccuracies may grow and overwhelm the final result. However, the role of rounding error in many algorithms is well understood.

The IEEE standard specifies two special numbers: an infinity to represent overflow and a not-a-number (NaN) for operations whose result is undefined. However, when such numbers are produced (e.g., when a product overflows) usually the right thing to do is terminate the calculation. For underflow the standard requires a controversial procedure called gradual underflow. In many cases, a calculation can be scaled so that overflows never occur and underflows are harmless.

—

2.4.1. Floating-point numbers

One of the problems with integers — or **fixed-point numbers** as they are sometimes called — is their limited range. For example, a 32-bit signed magnitude integer can only represent numbers between $\pm 214{,}477{,}483{,}648$. To represent a larger number, you would have scale the number by shifting it right to fit into the word (thereby losing some low order bits of precision) and then record the scaling factor. Such scaling is tedious and

error prone. **Floating-point** numbers solve this problem by building a scaling factor into the representation.

Floating-point numbers are analogues of the scientific notation that one encounters in the displays of calculators. Specifically, a calculator might display the number

$$-3.76542\ -03 \tag{2.6}$$

which stands for

$$-3.76542 \cdot 10^{-03} = -0.00376542.$$

The advantage of computing in such a system, as we have suggested, is that one can work with a wide range of numbers — in this case from about 10^{-100} to 10^{100} — without having to worry about scaling. The disadvantage is that some of the bits in the word containing the number must be used to represent the exponent, thus decreasing the precision of the representation as a whole.

Floating-point numbers have their special terminology. The **base** of the number (2.6) is 10. The number -03 is called the **exponent**. The number -3.76502 is called the **significand** (formerly the mantissa or fraction).[3]

Most floating-point systems impose a **normalization** on their numbers that insures that the significand is used to maximal effect. For example, in the six-digit floating-point system illustrated by (2.6) it would be pointless to represent 0.00376542 by

$$-0.03765\ -01,$$

since the two initial zeros cause a loss of accuracy of two figures in the significand. Thus most systems require that the leading digit of the significand be nonzero. (For an exception see the discussion of gradual underflow at the end of this subsection.)

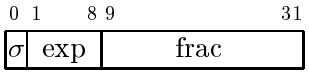
2.4.2. IEEE floating-point format

There are many decisions to be made in turning these general observations into a working floating-point system; and the early years of scientific computing were characterized by an embarrassment of such systems, each differing subtly from one another. Although programs written in a high-level language for one system would for the most part work on another, there were enough exceptions to make standardization desirable. The result

[3] This ill-formed term obviously derives by analogy with addend, multiplicand, dividend, and the like. These arithmetical terms derive in turn from a Latin form, called the gerundive, which is passive in voice. Thus the addend is the thing to be added; the multiplicand is the thing to be multiplied; and the dividend is the thing to be divided. It follows that the significand is the thing to be signified — whatever that means.

of an effort in the late 1970's and early 1980's was the IEEE floating-point system, which has become the standard for today's computers.

IEEE floating-point numbers come in two sizes — 32 bit and 64 bit. The 32-bit or **single-precision** form is

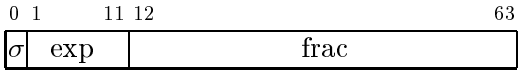


(2.7)

The small numbers above the box denote bit positions within the 32-bit word containing the number. The box labeled σ contains the sign of the significand. The other two boxes contain representations of the exponent and the significand. The value of the number is

$$(-1)^{\sigma}\ 1.\text{frac} \cdot 2^{\text{exp}-127}, \qquad 1 \leq \text{exp} \leq 254.$$

The 64-bit or **double-precision** form is



The value of a double precision number is

$$(-1)^{\sigma}\ 1.\text{frac} \cdot 2^{\text{exp}-1023}, \qquad 1 \leq \text{exp} \leq 2046.$$

Since the leading bit in the significand of a normalized, binary, floating-point number is always one, it is wasteful to devote a bit to its representation. To conserve precision, the IEEE stores only the part below the leading bit — the fraction — and recovers the significand via the formula $m = (-1)^{\sigma} \cdot 1.\text{frac}$.

The exponent field also contains a coded form of the exponent. The number exp is called a **biased exponent**, since the true value e of the exponent is computed by subtracting a bias. The unbiased exponent range for single precision is $[-126, 127]$, which represents a range of numbers from roughly 10^{-38} to 10^{38}. The unbiased exponent for double precision is $[-1022, 1023]$ — a range of numbers from roughly 10^{-307} to 10^{307}. In both precisions the extreme values of exp (i.e., 0 and 255 in single precision and 0 and 2047 in double) are reserved for special purposes.

Zero is represented by exp = 0 (one of the reserved exponents) and a significand of zero. The sign bit can be either 0 or 1, so that the system has both a +0 and a −0.

2.4.3. Rounding

In any nondegenerate interval there are an infinite number of real numbers, while only a finite number of floating-point can lie within the same interval. Thus real numbers must

be approximated by floating-point numbers. The process by which this approximation is effected is called **rounding.** The IEEE standard specifies four ways of rounding numbers, of which by far the most widely used is **round to nearest.** Specifically, a real number within the range of the floating-pont system is approximated by the nearest floating-point number. In case of a tie, the number is rounded to the nearest floating-point number whose least significant bit is zero.

The error made in rounding a number is measured in terms of **relative error**. Specifically, let $\mathrm{fl}(x)$ be the result of rounding x. Then the relative error in $\mathrm{fl}(x)$ is

$$\rho(x) = \frac{|\mathrm{fl}(x) - x|}{|x|}. \tag{2.8}$$

To see what this means, consider the following table in which $e = 2.71828182\ldots$ is rounded in decimal to successively higher precision.

Approximation	ρ
3.	$1 \cdot 10^{-1}$
2.7	$7 \cdot 10^{-3}$
2.72	$6 \cdot 10^{-4}$
2.718	$1 \cdot 10^{-4}$
2.7183	$7 \cdot 10^{-6}$
2.71828	$7 \cdot 10^{-7}$

These results suggest the following rule of thumb: If the relative error in x is approximately 10^t then x and its approximation agree to about t decimal digits.

All this shows that bounds on $\rho(x)$ in (2.8) will tell us a great deal about the effects of rounding in the IEEE system. Such bounds can be written in terms of the distance between 1 and the next larger floating-point number. This number is called **machine epsilon** or the **rounding unit** and is frequently written **u**. In this notation, for round to nearest we have

$$\rho(x) \leq \frac{1}{2}\mathbf{u}.$$

Equivalently this can be written in the form

$$\mathrm{fl}(x) = x(1+\epsilon), \quad \text{where } |\epsilon| \leq \frac{1}{2}\mathbf{u}.$$

Thus rounding to nearest amounts to multiplying by a number that deviates from one by a quantity not greater than half the rounding unit.

For single precision the value of **u** is

$$\mathbf{u} = 2^{-23} \cong 1.2 \cdot 10^{-7},$$

and for double precision

$$\mathbf{u} = 2^{-52} \cong 2.2 \cdot 10^{-16}.$$

Thus rounding to single precision gives gives about 7 accurate digits. Rounding to double precision gives about 16 accurate digits.

The basic arithmetic operations in any floating-point system are addition, subtraction, multiplication, and division (and sometimes the square root). The IEEE system specifies that these operations must return the correctly rounded result (provided they are within the range of normalized floating-point numbers). In particular, if the rounding mode is round to nearest, then for $\circ = +, -, \times, /$ the floating-point operation $\mathrm{fl}(a \circ b)$ must satisfy

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \epsilon),$$

where

$$|\epsilon| \leq \frac{1}{2}\mathbf{u}.$$

An important consequence of rounding error is that the result of evaluating an expression can depend on the order in which it is evaluated. For example, if we code

```
sum1 = a + (b + c)
sum2 = (a + b) + c
```

`sum1` and `sum2` may not have the same value. The difference can be quite dramatic. For example, consider the sum

$$\begin{array}{rr} & 472635.0000 \\ + & 27.5013 \\ - & 472630.0000 \\ \hline & 32.5013 \end{array}$$

If we compute this sum in the order given in six digit arithmetic, we get first

$$\begin{array}{rr} & 472635.0000 \\ + & 27.5013 \\ \hline & 472663.0000 \end{array}$$

and then

$$\begin{array}{rr} & 472663. \\ - & 472630. \\ \hline & 33. \end{array}$$

which is accurate to only two digits. On the other hand if we compute

$$\begin{array}{r} 472635. \\ -\ 472630. \\ \hline 5. \end{array}$$

and then

$$\begin{array}{r} 5.0000 \\ +\ 27.5013 \\ \hline 32.5013 \end{array}$$

which is exact.

A treatment of rounding error and its effects is beyond the scope of this book, and the reader should refer to one of the many excellent treatments of this topic. It is important not to let examples like the above induce an unwarranted fear of rounding error. There are two reasons. First, for many algorithms the effects of rounding error have been analyzed and are well understood. For example, the above sum is extremely sensitive to perturbations in its larger terms — the technical term is that it is **ill-conditioned** — and the rounding error merely unmasks this sensitivity. Second, when analysis is lacking, empirical studies and experience often show that the ill effects of rounding error are limited. The best advice is to be informed, proceed cautiously, but keep computing.

2.4.4. Floating-point exceptions

Like integers, floating-point numbers are not closed under arithmetic operations. An example (which also applies to the real numbers) is the quotient $x/0$, where x is nonzero. This division by zero is called a **divide-by-zero exception.** The IEEE standard requires that exceptions be signaled by setting a special flag that can be interrogated by the programmer. It recommends that the programmer be able to enable an interrupt on exceptions.

The response to an exception varies with the kind of exception. The response to divide by zero is to create a number called an **infinity.** An infinity, has exp $= 255$ and frac $= 0$ in single precision [see (2.7)], with an analogous representation in double precision. Like the IEEE zero, infinity can have two signs. The sign is determined as you might expect. If the numerator and the denominator (± 0) have the same sign then the sign of the resulting infinity is plus; otherwise it is minus. Operations with infinities are defined also in a natural manner; e.g., $\infty \times x = \infty$, provided $x > 0$.

Some exceptions result from invalid operations. For example, there is no convenient definition for $0/0$ or $\infty - \infty$. When such invalid operations occur, the result is given the value **NaN (Not a Number),** which in single precision is represented by exp $= 255$ and frac $\neq 0$. Operations involving a NaN produce a NaN.

When a single precision operation would result in a value of exp that is greater than 254, it is said to have **overflowed.** IEEE standard specifies that (in round to nearest mode) an overflow produces and infinity of appropriate sign. However, an overflow is almost always a fatal error, and most systems trap on overflow and stop the process in question.

When a floating-point operation would produce a value of exp that is less than 1, it is said to have **underflowed.** Underflow is often not a fatal error, and a natural response to underflow is to treat the result as a zero (called **flush to zero**). To see why consider the following example, in which we use decimal floating point with a six digit significand and an exponent range of $[-99, 99]$.

Given two floating-point numbers a and b it is desired to compute

$$c = \sqrt{a^2 + b^2}. \tag{2.9}$$

This is a potentially dangerous computation. If either a or b is greater than 10^{49}, overflow occurs when it is squared, even if the final result can be represented, e.g, when $a = b = 10^{60}$ and therefore $c = \sqrt{2}\cdot 10^{60}$.

We can cure this problem by the following algorithm.

```
1.  s = abs(a) + abs(b);
2.  c = s*sqrt((a/s)^2 + (b/s)^2);
```
(2.10)

Mathematically, this algorithm gives the same result as (2.9). But in our example, where $a = b = 10^{60}$, we compute

$$\sqrt{2} = 2\cdot 10^{60}\sqrt{(\tfrac{1}{2})^2 + (\tfrac{1}{2})^2}$$

with no overflows.

However the algorithm (2.10) fails when $a = 10^{60}$ and $b = 1$, in which case the correctly rounded value of c is 10^{60}. The reason is that the computed value of s is 10^{60}, and the algorithm underflows when it computes $(b/s)^2$. On the other hand, if $(b/s)^2$ is flushed to zero, it is easy to see that the result is 10^{60}, which is the correctly rounded answer.

The above example is not an uncommon case. Many kinds of floating point computations can be scaled so that overflows to not occur and underflows are harmless provided they are flushed to zero.

The IEEE standard requires a more elaborate response to underflow called **gradual underflow,** which sets exp to zero and treats frac as a binary fraction with the binary point at the left. The value of this **subnormal number** is frac$\cdot 2^{-126}$. It is easier to see what is going on here in a decimal system. The following table shows the representation

of $\pi \cdot 10^k$ as k approaches the underflow point.

k	representation
-98	3.14159 -98
-99	3.14159 -99
-100	0.31416 -99
-101	0.03142 -99
-102	0.00314 -99
-103	0.00031 -99
-104	0.00003 -99
-105	0.00000 00

Thus gradual underflow provides a staged intermediary between the smallest fully representable number and zero. However, as we proceed our numbers loose significant digits.

Gradual underflow is highly controversial. It solves certain technical problems in designing numerical algorithms. On the other hand, it adds to the complexity of the floating-point system, and in pipelined implementations (to be discussed in §3.4) it can slow down the computation. Consequently, some manufacturers provide only software support for gradual underflow. This works well when underflows are infrequent; but when they are not, the software response will again slow down the computation. There does not seem to be any easy solution to this dilemma.

2.5. Operation counts

Synopsis: One way to compare algorithms that solve the same problem is to count the number of floating-point operations they perform. Although such counts underestimate the execution time, that time is frequently proportional to the count. A count can often be divided into two factors — a function of the size of the problem, called the order, and a constant, called the order constant. In comparing two algorithms of the same order, one must examine the order constant. For constants of different order, the one with the higher order will ultimately be faster. But ultimately may never come in practice.

For computations that take place in loops there is an integration technique that often simplifies the drudgery of performing operation counts.

—

It frequently happens that a problem can be solved in more than one way. Deciding which of two algorithms to use involves balancing a number of factors — for example, numerical stability, suitability to the problem at hand, parallelizabilty, and execution time. Usually, the last item, execution time, must be determined empirically. But in comparing two algorithms for which floating-point operations are the chief computa-

tional burden, there is a short cut: count the number of arithmetic operations performed by each. Such a count is called an **operation count.**

Consider, for example, the problem of computing the matrix-vector product $y = Ux$, where U is an upper triangular matrix of order n. Writing out the product elementwise, we have

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1,n-1} & u_{1n} \\ 0 & u_{22} & \cdots & u_{2,n-1} & u_{1n} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & \cdots & 0 & u_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}.$$

From this we see that y_i is given by

$$\begin{aligned} y_i &= 0\cdot x_1 + 0\cdot x_1 + \cdots + 0\cdot x_{i-1} + u_{ii}x_i + \cdots + u_{in}x_n \\ &= u_{ii}x_i + \cdots + u_{in}x_n. \end{aligned}$$

Hence

$$y_i = \sum_{j=i}^{n} u_{ij}x_j.$$

The following code fragment implements this formula.

```
1.  for i=1:n
2.     y(i) = 0;
3.     for j=i:n
4.        y(i) = y(i) + U(i,j)*x(j);
5.     end
6.  end
```
(2.11)

To get an operation count for (2.11), we observe that the body of the inner loop on `i` (line 4) contains one addition. Consequently, the this loop performs

$$\sum_{j=i}^{n} 1 = n - i + 1$$

additions. Since the outer loop on `i` is executed `n` times, the total number of additions is

$$\sum_{i=1}^{n} n - i + 1 = n^2 - \frac{n(n+1)}{2} + n = \frac{n^2}{2} + \frac{n}{2}. \tag{2.12}$$

Here we have used the well-known formula $\sum_{i=1}^{n} i = n(n+1)/2$. The number of multiplications is the same is the same as the number of additions.

As n becomes large, the term n^2 in the count (2.12) dominates the term $n/2$. For example, when $n = 10$, the second term is one-tenth the first. It is therefore customary to give only the highest order term in an operation count — in our example $n^2/2$. This term has two parts. The first is the function n^2, which is called the **order** of the count and is written $O(n^2)$. The second is the constant $\frac{1}{2}$, which is called the **order constant**.

Operation counts are not useful in predicting actual execution times. The reason is that there is much more going on in a program like (2.11) than floating-point operations. For example, there is overhead in incrementing and testing `j` in the inner loop. Again, the position of `U(i,j)` in memory must be calculated before it can be retrieved. Consequently, multiplying the count $n^2/2$ by the sum of the times for a floating-point add and multiply will underestimate the time it takes to execute the algorithm.

In many algorithms, however, the additional overhead is proportional to the number of floating-point operations. When this is true, operation counts can be used to make rough comparisons between algorithms. There are two cases to consider.

The first is when the the algorithms are of the same order. It is then the order constant that determines which algorithm is the faster. But this statement must be qualified. The overhead for the operation counts may be different for the two algorithms. This means that an algorithm with order constant two may be faster than one with order constant one. However, when the order constants differ greatly, say one to ten, it is wise to bet on the algorithm with the smaller operation count.

The second case is when the orders differ, say n^2 versus $n \log n$ (as is the case with the Fourier transform versus the fast Fourier transform). This means that overheads being equal, the algorithm with the lower order will, in the long run, outstrip the one with the higher order. For example, when $n = 10^6$, $n^2/n \log n = 7.2{\cdot}10^4$, so that the speedup may be as large as four orders of magnitude. But, as John Maynard Keynes observed, in the long run we are all dead. If the algorithm with the lower order has a very large order constant, the one with the higher order may be preferable. And it should be kept in mind that a numerically unstable algorithm that returns the wrong answer is useless, however low its order count.

As we have noted it is customary to report only the high order term in an operation count. In some cases this term can be obtained by integration, without recourse to complicated summation formulas. Specifically, in our running example we approximate the sum $\sum_{i=1}^{n} \sum_{j=i}^{n} 1$ by

$$\int_0^n \int_i^n 1 \, dj \, di = \frac{n^2}{2}.$$

Note that in the integral we have adjusted the limits of summation by a subtraction

of one to make the integral have no terms of order n or 1. This trick of substituting integrals for summation has wide applicability in computing operations counts.

EXERCISES

1. Find an extended ASCII table on the web and answer the following questions.

    1. Give the binary, hex, and decimal equivalents of 'a', 'b', 'z', 'A', 'B', 'Z'.
    2. Give an arithmetic algorithm for converting lower case ASCII to upper case ASCII.

2. The numbers 0–9 are represented by the ASCII codes x30–x39. Speculate on why they do not have their natural values x00–x09.

♠ Alphabetization problem.

3. Write a matlab function `isdigit(c)` that returns `1` if its argument is `'0',..., '9'` and otherwise returns `0`. Note that the specification is very strict: `isdigit(2)` and `isdigit('02')` should return zero. [Hint: You can do it using the Matlab functions `ischar`, `size`, and `str2num`.]

4. Shifting the bits of an unsigned integer left or right is equivalent to respectively multiplying or dividing the integer by a power of two. If the quotient is not an integer it is rounded toward zero. This process does not work for signed magnitude representation of signed integers. Devise **arithmetic right and left shifts** that solve this problem.

5. In the table (2.5) the last three bits in the 2's complement representations of -1 to -8 are the representations of 7 to 0. Give a mathematical argument to show why this happens.

6. In 2's complement arithmetic, an arithmetic right shift is usually defined by propagating the sign bit into the number, as illustrated below for a five digit number.

$$10101 \longrightarrow 11010 \longrightarrow 11101 \longrightarrow \cdots .$$

Show that each such shift divides the integer by two, but any rounding is toward $-\infty$.

7. Write a matlab function with the following specifications.

```
function bs = dec2bin(d, nbits)

% DEC2BIN takes a nonnegative floating point number d and produces its
% binary representation as a string of zeros and ones.
% If the fractional part of d is nonzero it is separated from the
% integer part by a binary point.  The optional argument nbits (default
% 45) is the maximum number of bits to use in the representation.  If it
% is exceeded before the integer part is computed the string is preceded
% by '...'.  If it exceeded before the fractional part has been
% computed the string is followed by '...'.
```

8. Determine to several decimal places the largest number that can be represented in IEEE double precision. Do the same for the smallest unnormalized number.

9. The following Matlab code implements Gaussian elimination on a matrix `A` of order `n`.

```
1.  for k=1:n-1;
2.     for i=k+1:n;
3.        A(i,k) = A(i,k)/A(k,k);
4.     end
5.     for j=k+1:n
6.        for i=k+1:n
7.           A(i,j) = A(i,j) - A(i,k)*A(k,j);
8.        end
9.     end
10. end
```

Determine up to lower order terms the number of additions (a subtraction counts as an add), multiplications, and divisions performed by the algorithm.

10. Floating point multiplications usually considerably faster than floating-point divisions. Consequently, one might be tempted to replace the loop in lines 2–4 by the following code.

```
temp = 1/A(k,k);
for i=k+1:n
   A(i,k) = temp*A(i,k);
end
```

Why do most codes not bother with this "improvement?"