Chapter 2
Computer Numbers and MATLAB

2.1 Introduction

In Section 1.2 on matrix inversion the differences between computation with numerical algorithms and exact mathematical symbolic manipulation became clear. Virtually all mathematical computations suffer the same fate – accumulated error – when executed on computers. On most computers, numerical algorithms are implemented using the *binary* number system, 0's and 1's, leading to results that are very accurate but not exact. In this chapter we will see how numbers are represented by computers. In later chapters, we will see some of the implications of the inaccuracies that accompany computer representation of numbers.

Number systems are *positional* with standard conventions. For example, the decimal or base 10 number 27.75 means

$$2 \cdot 10^1 + 7 \cdot 10^0 + 7 \cdot 10^{-1} + 5 \cdot 10^{-2} = 2 \cdot 10 + 7 \cdot 1 + \frac{7}{10} + \frac{5}{100}.$$

Positional number systems require a positive base and a set of symbols. In the decimal system the base is 10 with symbols 0, 1, 2, ... 8, 9. The binary number system uses a base of 2 with symbols 0 and 1. Other common number systems are *octal*, base 8, and *hexadecimal*, base 16. In general terms, we may represent a base b number as

$$a_n b^n + a_{n-1} b^{n-1} + \ldots + a_2 b^2 + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + \ldots$$

$$= (a_n a_{n-1} \ldots a_2 a_1 a_0 . a_{-1} a_{-2} \ldots)_b$$

where the *radix* point, . , between a_0 and a_{-1}, divides the number into integral and fractional parts. The a_k's are specified by the symbol set associated with the base b.

Convince yourself that each of the following numbers represent the same value:

$$(1001.01)_2, \quad (11.2)_8, \quad (9.25)_{10}, \quad (9.4)_{16}$$

In general, the binary representation of a real number will be the longest.

2.2 Binary Numbers

A binary number will have the general form

$$a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_2 2^2 + a_1 2^1 + a_0 2^0 + a_{-1} 2^{-1} + \ldots$$

$$= (a_n a_{n-1} \ldots a_2 a_1 a_0 . a_{-1} a_{-2} \ldots)_2,$$

where each a_k is either a zero or one. Note that subscripts on the a_k's correspond to the powers of 2.

Computations with binary numbers often involve two special binary integers. The first is a binary integer consisting of m bits (0's or 1's) with the form $(100\ldots000)_2$, in other words, a leading 1 followed by $m-1$ zeros. Convince yourself that the value of $(100\ldots000)_2$ is equal to $1\cdot 2^{m-1}$. Remember that the zero on the right is associated with 2^0. The second binary number is more involved representing an integer m bits long of the form $(111\ldots111)_2$, a row of m 1's. In powers of 2, $(111\ldots111)_2$ becomes

$$1\cdot 2^{m-1}+1\cdot 2^{m-2}+\ldots+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0.$$

In reverse order we recognize a finite *geometric* series with a constant ratio of 2.

$$2^0+2^1+2^2+\ldots+2^{m-2}+2^{m-1}.$$

All calculus textbooks describe how to sum a geometric series. In this case the sum of the series is 2^m-1. This result is used often in computations with binary numbers. For example, the binary number $(11111)_2=1+2+4+8+16=2^5-1=31$.

A common task is to convert a decimal number to binary. Suppose we have a decimal value in the form $(x.y)_{10}$, for example $(28.75)_{10}$, and wish to determine the binary equivalent. The standard approach is to treat the integral and fractional parts separately. We begin by equating the integral part of both representations

$$x=a_n2^n+a_{n-1}2^{n-1}+\cdots+a_22^2+a_12^1+a_02^0. \tag{2.1}$$

Dividing a decimal integer, x, by 2 will result in a remainder of either zero or one. In other words, either the integer is divisible by 2 (remainder is zero) or it is not divisible by 2 (remainder is one). Dividing equation (2.1) by 2 gives

$$x/2=a_n2^{n-1}+a_{n-1}2^{n-2}+\cdots+a_22^1+a_1+a_0/2=q+r/2.$$

The remainder term, $r/2$ equals $a_0/2$; thus, r is equal to a_0 which is either a zero or a one. This scheme may be repeated using the integral value q. Form the quotient $q/2$ to determine a_1, and so forth. The method terminates when the quotient becomes zero. An example will demonstrate the procedure.

Example: Convert the decimal integer 28 to binary.

$$\begin{aligned}
28/2&=14+0/2 &&\Rightarrow a_0=0\\
14/2&=7+0/2 &&\Rightarrow a_1=0\\
7/2&=3+1/2 &&\Rightarrow a_2=1\\
3/2&=1+1/2 &&\Rightarrow a_4=1\\
1/2&=0+1/2 &&\Rightarrow a_5=1
\end{aligned}$$

The results show that the binary form of $(28)_{10}$ is

$$(11100)_2=1\cdot 2^4+1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+0\cdot 2^0=16+8+4.$$

A word of caution with this scheme. The binary coefficients are produced beginning with a_0 followed by a_1, a_2, etc. Convert 44 to binary and check your results with MATLAB's **dec2bin**.

>> dec2bin(44)
ans =
101100

Representing the fractional part of the decimal value $(x.y)_{10}$ calls for a different approach. Multiplying a decimal fraction, **.***y*, by 2 will result in an integral part of either zero or one. Multiply .3 by $2 = 0.6$ or .7 by $2 = 1.4$. Equating the fractional parts of the decimal and binary representations gives

$$.y = a_{-1}2^{-1} + a_{-1}2^{-2} + a_{-3}2^{-3} + \ldots. \tag{2.2}$$

Multiplying both sides of equation (2.2) by 2 gives

$$2(.y) = 2(a_{-1}2^{-1} + a_{-1}2^{-2} + a_{-3}2^{-3} + \ldots)$$

or
$$2(.y) = a_{-1} + a_{-2}2^{-1} + a_{-3}2^{-2} + \ldots. \tag{2.3}$$

Note that, if $.y \geq 0.5$, a_{-1} will be a one, otherwise a zero. Now, subtract a_{-1} from both sides of (2.3) and again multiply by 2.

$$2[2(.y) - a_{-1}] = 2(a_{-2}2^{-1} + a_{-3}2^{-2} + \ldots) = a_{-2} + a_{-3}2^{-1} + \ldots.$$

a_{-2} will be either zero or one. The scheme continues until either it terminates with a finite number of multiplications or else gives an endless sequence of zeros and ones.

Example: Convert $(0.75)_{10}$ to binary.

$$\begin{aligned} 2(0.75) = 1.5 &\quad \Rightarrow a_{-1} = 1 \\ 2(1.5 - 1) = 1.0 &\quad \Rightarrow a_{-2} = 1 \\ 2(1.0 - 1) = 0.0 &\quad \Rightarrow a_{-3} = 0 \end{aligned}$$

At this point the scheme terminates as all further coefficients will be zero. Thus,

$$(.75)_{10} = (.11)_2 = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 0.5 + 0.25.$$

As an alternative method, a decimal fraction such as 0.75 may be multiplied by a power of 2 to produce an integer. In this case $0.75 = 3/4$, so that $(.75) \cdot 2^2 = 3$. The binary representation of 3 is $(11)_2$. We now multiply by 2^{-2} and shift the binary point.

$$(.75)_{10} = (3)_{10} \cdot 2^{-2} = (11)_2 \cdot 2^{-2} = (.11)_2.$$

This scheme will work only if the fractional part, **.**y, can be expressed as a rational number with a denominator that is a power of two. The method will not work on a fraction such as $0.9 = 1.8/2 = 3.6/4 = 7.2/8$, etc., since the numerator will never be an integer.

To represent our example value $(28.75)_{10}$ as a binary number, the integral and fractional parts may be combined to give $(28.75)_{10} = (11100.11)_2$.

A minor change in our example number, from $(28.75)_{10}$ to $(28.9)_{10}$, complicates the issue. The integral part remains unchanged, $(11100)_2$; however, converting the fractional part, $(.9)_{10}$, to binary reveals a never ending sequence. Using the procedure developed earlier:

$$\begin{aligned} 2(0.9) = 1.8 &\quad \Rightarrow a_{-1} = 1 \\ 2(1.8 - 1) = 1.6 &\quad \Rightarrow a_{-2} = 1 \\ 2(1.6 - 1) = 1.2 &\quad \Rightarrow a_{-3} = 1 \\ 2(1.2 - 1) = 0.4 &\quad \Rightarrow a_{-4} = 0 \\ 2(0.4 - 0) = 0.8 &\quad \Rightarrow a_{-5} = 0 \\ 2(0.8 - 0) = 1.6 &\quad \Rightarrow a_{-6} = 1 \end{aligned}$$

At this point we observe that a_{-6} is identical to a_{-2}, and so a_{-7} will be identical to a_{-3}, a_{-8} will be identical to a_{-4} and so forth resulting in a repeating, non-terminating binary fraction. In other words, $(28.9)_{10}$ has an infinite binary representation:

$$(28.9)_{10} = (11100.1\,1100\,1100\,1100\;1100\ldots)_2.$$

Obviously, the non-terminating representation of $(28.9)_{10}$ cannot be stored in a binary computer.

Although our efforts have been focused on converting decimal numbers to their equivalent binary representations, it should be clear that the methods and procedures discussed may be applied to other bases. For example, a decimal integer may be converted to an octal integer, base 8, by dividing by 8 and following the method described for binary numbers. The octal system uses the symbols $0,\ 1,\ 2,\ \ldots\ , 7$.

Example: Convert the decimal number 135.45 to octal. We begin with the integral part.

$$\begin{aligned} 135/8 = 16 + 7/8 &\quad \Rightarrow a_0 = 7 \\ 16/8 = 2 + 0/8 &\quad \Rightarrow a_1 = 0 \\ 2/8 = 0 + 2/8 &\quad \Rightarrow a_2 = 2 \end{aligned}$$

Conversion of the fractional part will involve multiplication by 8.

$$\begin{aligned} 8(0.45) = 3.6 &\quad \Rightarrow a_{-1} = 3 \\ 8(3.6 - 3) = 4.8 &\quad \Rightarrow a_{-2} = 4 \\ 8(4.8 - 4) = 6.4 &\quad \Rightarrow a_{-3} = 6 \\ 8(6.4 - 6) = 3.2 &\quad \Rightarrow a_{-4} = 3 \\ 8(3.2 - 3) = 1.6 &\quad \Rightarrow a_{-5} = 1 \\ 8(1.6 - 1) = 4.8 &\quad \Rightarrow a_{-6} = 4 \end{aligned}$$

The values in the table reveal a repeating sequence, 4631. Combining the integral and fractional parts gives

$$(135.45)_{10} = (207.3\,4631\,4631\,4631\;4631\ldots)_8.$$

2.3 Computer Representation

Consider the task of approximating the binary equivalent of 28.9 within a computer. Aside from the problem of an infinite number of binary digits in the number there are other issues. The representation of $(28.9)_{10}$, in the last section,

$$(11100.1\,1100\,1100\,1100\,1100\ldots)_2$$

may be written in *floating-point* form by moving the binary point four places to the left so that there is a leading one to the left of the binary point.

$$(1.11001\,1100\,1100\,1100\,1100\ldots)_2 \cdot 2^4.$$

The last form suggests a standard structure for a real binary floating-point number,

$$\text{R} = (\text{sign}) \cdot (1.b_1b_2b_3b_4b_5b_6\ldots)_2 \cdot 2^{(\text{exponent data})}.$$

The term $(1.b_1b_2b_3b_4b_5b_6\ldots)_2$ is typically called the *mantissa*. The smallest mantissa is $(1.00000\ldots)_2 = 1$ and the largest is $(1.11111\ldots)_2 = 2$. To store the number R, a computer must save information about the sign, the fractional part of the mantissa and the exponent data. Clearly, only a finite number of zeros or ones is available to represent the number. Standards in the computer industry dictate the techniques and procedures used to store R.

MATLAB uses the IEEE standard for double precision floating-point numbers. In brief, the standard calls for a 64 bit *word* to represent R with the following details: one bit for the sign of R (usually zero indicates a positive value), 11 bits for the exponent data, and 52 bits for the fractional part of the mantissa (the leading one is not stored). With this 52 bit restriction the fractional part of the mantissa is bounded by $(.00000\ldots0)_2 = 0$ and $(.11111\ldots1)_2 = 1 - 2^{-52}$, a value very close to one. We have

$$(.00000\ldots0)_2 \le (.b_1b_2b_3b_4b_5b_6\ldots b_{52})_2 \le (.11111\ldots1)_2$$

or

$$0 \le (.b_1b_2b_3b_4b_5b_6\ldots b_{52})_2 \le 1 - 2^{-52} \approx 1.$$

The method used to store the exponent data requires further explanation. The 11 bits allocated for exponent data permit storage of eleven zeros and, at the other extreme, eleven ones, $(11111111111)_2 = 2^{11} - 1 = 2047$. In other words, all non-negative integer values between 0 and 2047 may be stored in the 11 bits. These 2048 numbers are used to represent both positive and negative exponents. This is accomplished by storing a signed exponent plus a constant in the eleven bits assigned to the exponent data. The constant, usually called the *bias*, is $1023 = (01111111111)_2$. As an example, if the exponent is -45 we actually store the positive binary integer equal to $-45 + 1023$ or 978 in the eleven bits. Since

```
>> dec2bin(978)
ans =
1111010010
```

the eleven bit string 01111010010 is stored to represent -45.

To summarize our work so far consider an example 64 bit IEEE binary word usually called a *machine number*

$$1\;01111111000\;1110001000\;0000000000\;0000000000\;0000000000\;0000000000\;00.$$

With proper interpretation, the machine number may be converted to a decimal value. First, the one in the leftmost position indicates that the number is negative. Second, the exponent data is specified by the next eleven bits 01111111000 which corresponds to the decimal integer

```
>> bin2dec('01111111000')
ans =
     1016
```

Using a bias of 1023, the actual base 2 exponent is $1016 - 1023 = -7$. Third, using the remaining 52 bits with trailing zeros deleted, the mantissa is given by 1.1110001. Recall, the leading one is not stored. Combining terms, the machine number becomes

$$-(1.1110001)_2 \cdot 2^{-7}$$

which equals $$-(1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-7}) \cdot 2^{-7},$$

or $$-(2^{-7} + 2^{-8} + 2^{-9} + 2^{-10} + 2^{-14}).$$

Finally, the exact decimal value of the machine number is

$$-0.01470947265625.$$

Continuing our analysis, we note that exponent data is bounded by the extreme values.

$$0 \leq \text{exponent} + \text{bias} \leq 2047$$

or $$0 \leq \text{exponent} + 1023 \leq 2047.$$

Conventional use reserves the extreme values 0 and 2047 for special cases leading to

$$1 \leq \text{exponent} + 1023 \leq 2046.$$

Thus the limits on the actual base 2 exponent become

$$-1022 \leq \text{exponent} \leq 1023.$$

In other words, the largest positive real number that can be represented using the IEEE standard is $(1.1111\ldots1111)_2 \cdot 2^{1023}$. This equals MATLAB's **realmax**.

```
>> realmax
ans =
  1.7977e+308
```

To appreciate the magnitude of **realmax**, recall that *Avogadro's* number is $6.023 \cdot 10^{23}$ molecules/mole. Clearly, $(1.1111\ldots1111)_2 \cdot 2^{1023}$ is very close to 2^{1024}. If we attempt to compute 2^{1024} with MATLAB we find

```
>> 2^1024
ans =
   Inf
```

where **Inf** is the IEEE representation for positive infinity, an overflow in this case. In other words, 2^{1024} cannot be represented by the 64 bit word described above.

To bring our discussion of computer numbers to a close, it is important to understand how the IEEE standard influences accuracy. The length of the fractional part of the mantissa, 52 bits, given by $(.b_1b_2b_3b_4b_5b_6\ldots b_{52})_2$, is the critical item. Using the standard structure for binary floating-point numbers, it should be clear that all integers less than $(1.1111\ldots 111)_2 \cdot 2^{52}$ can be stored exactly. The exponent value of 52 will shift the binary point 52 places to the right giving a binary integer with 53 ones which equals $2^{53}-1$. Actually, it is possible to store 2^{53} exactly.

```
>> format long e
>> (2^53)-1
ans =
   9.007199254740991e+015
>> 2^53
ans =
   9.007199254740992e+015
```

However, it is not possible to store $2^{53}+1$ exactly.

```
>> (2^53) +1
ans =
   9.007199254740992e+015
```

The value of 2^{53}, 9007199254740992, is a 16 digit integer. This result indicates that all 15 digit decimal integers and most 16 digit decimal integers will store exactly in the 52 bits allocated. In summary, computations with the IEEE 64 bit word are accurate to 15 decimal places. Note that the **long e** format displays 15 decimal places.

2.4 Problems

2-1. Convert the following numbers to binary, octal and hexadecimal. The hexadecimal system, base 16, uses the symbols 0, 1, 2, $\cdots$, 9, A, B, C, D, E, F where A corresponds to 10, B corresponds to 11 and so forth. Show your work.
a) $(127)_{10}$, b) $(0.0625)_{10}$, c) $(43.68)_{10}$

2-2. One octal digit corresponds to 3 binary digits and one hexadecimal digit corresponds to 4 binary digits. Why? Convert the binary number 101011.101011100011 to both octal and hexadecimal by grouping the bits in threes, $101\,011.101\,011\,100\,011$, and then in fours, $0010\,1011.1010\,1110\,0011$, moving both ways from the binary point.

2-3. Convert the decimal fraction 0.01470947265625 to binary. Check your answers with the results in the text.

2-4. Determine the 64 bit machine number that represents $(28.75)_{10} = (11100.11)_2$.

2-5. Consider a computer with a 32 bit word as follows: one bit for the sign of R, 8 bits to represent exponent data, and 23 bits for the fractional part of the mantissa. The bias is $(01111111)_2$. See Section 2.3.

a. What is the range of exponent data?
b. Determine the largest positive integer that may be stored.
c. Discuss the accuracy of this computer.

2-6. Consider the Ho-Hum computer with a 16 bit word as follows: bit number 1 stores the sign of R (0 for positive); bits number 2 - 6 the exponent data, and bits 7 - 16 the fractional part of the mantissa. The bias is $(01111)_2$. See Section 2.3.

a. What is the range of exponent data?
b. Determine the largest positive integer that may be stored.
c. Discuss the accuracy of the Ho-Hum computer.
d. Determine the decimal equivalent of the machine number

$$0\ 11000\ 1000100001$$

e. What is the next machine number after $0\ 11000\ 1000100001$?

2-7. Use MATLAB's **realmin** to determine the smallest positive real number that can be expressed. Explain the result using the IEEE 64 bit word.

2-8. The factorial function is a source of integers. Verify the statement found in >> **help factorial** by computing the factorials of 21, 22 and 23. It will be necessary to look up the factorials in mathematical tables.

2-9. The value >> **2^53** is exact. Show that >> **2^54** is not.