UNIVERSITY OF CALIFORNIA BERKELEY — Engineering 7
Department of Civil and Environmental Engineering
Spring 2013 — Professor: S. Govindjee

Internal representation of numbers

1 Introduction

In everyday life, numbers are almost exclusively represented using the base 10 decimal system. This means, for example, that an integer N written as

$$N = a_n a_{n-1} \cdots a_1 a_0 \tag{1}$$

corresponds to a short-hand for

$$N = (a_n a_{n-1} \cdots a_1 a_0)_{10} = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \cdots + a_1 \times 10^1 + a_0 \times 10^0 \tag{2}$$

As a concrete example we have that:

$$N = (231)_{10} = 2 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 \tag{3}$$

In the above case, 10 is called the *base* (or *radix*) of the system. The a_i's are integers between 0 and 9; i.e., $a_i \in \{0, 1, 2, \cdots, \text{base} - 1\}$.

Rational numbers less than 1, numbers with decimal points, make use of the negative powers of the base:

$$N = (0.a_1 a_2 \cdots a_{n-1} a_n)_{10} = a_1 \times 10^{-1} + a_2 \times 10^{-2} + \cdots + a_{n-1} \times 10^{-(n-1)} + a_n \times 10^{-n} \tag{4}$$

As a concrete example we have that:

$$N = (0.231)_{10} = 2 \times 10^{-1} + 3 \times 10^{-2} + 1 \times 10^{-3} \tag{5}$$

2 Binary Representation

Computers generally use a base 2 system which is called the *binary system.* An integer N is represented as

$$N = (a_n a_{n-1} \cdots a_1 a_0)_2 = a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_1 \times 2^1 + a_0 \times 2^0 \tag{6}$$

The a_i's are only allowed to be 0 or 1. As a concrete example we have:

$$N = (11100111)_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \tag{7}$$

The binary system is quite convenient to computers, since the basic unit of information that they store is an electrical charge which can be either "on" or "off". A single binary digit (a 0 or a 1) is called a *bit.* An 8-bit sequence of 0's and 1's is usually called a *byte.*

Determining the binary representation of integers is made easy in MATLAB by the command `dec2bin`. For example,

```
>> A=dec2bin(231)

A =

11100111
```

You can convert the other way using `bin2dec`.

2.1 Storage of binary integers

The set of all integers is countably infinite. This means that in order to represent all integers, computers would need infinite storage capacity. In reality, computers can store only a finite subset of all integers because each integer is only allowed a certain number of bits. The number of bits allocated for any integer stored is sometimes called a *word-length*. Using a 16-bit integer word-length, the integer N = 231 may be stored as

$$0\ 000000011100111\,. \tag{8}$$

The first bit, of the 16 bits, stores the sign of the number (here "positive" is stored as 0), while the remaining 15 bits store the number itself using the binary system representation.

Example 1 What is the largest integer that can be stored in a 32-bit signed integer? The binary representation for this would be the case where are the bits (save the sign bit) are set to unity:

$$0\ \underbrace{1111111111111111111111111111111}_{31\text{ bits}} \tag{9}$$

This is equivalent to

$$2^0 + 2^1 + 2^2 + \cdots + 2^{30} = \frac{1-2^{31}}{1-2} = 2^{31} - 1 \approx 2 \times 10^9 \tag{10}$$

or roughly 2 billion. For unsigned integers, 32-bits are available and one can get up to 4 billion. If you type `intmax` in MATLAB it will show you the precise value.

Example 2 Convert the integer $(578)_{10}$ to base 3. This can be performed by repeatedly subtracting successive powers of 3 from the number until one arrives at zero. We start the process with the largest power of 3 that will divide into $(578)_{10}$. This can be computed from $y = \lfloor \log_3(578) \rfloor$. We can now divide our number by 3^y to find out how many times it goes into $(578)_{10}$, and then subtract this amount and repeat. A simple bit of code that achieves this is

```
% base conversion
clear;
x = 578;
```

```
while x~=0
    y = floor(log(x)/log(3));  % Compute highest exponent
    d(y+1) = floor(x/3^y);     % Determine digit value
    x = x - d(y+1)*3^y;        % Deflate number
end
disp('Base 3 representation');
disp(d(end:-1:1));
```

The result is $(210102)_3$. Our algorithm can be checked using the built-in MATLAB function `dec2base(578,3)`.

2.2 Binary storage of real numbers

Real numbers, as opposed to integers, have decimal points. They can be represented in many different ways. One convenient way is to represent them in scientific notation or *floating-point* form:

$$x = \pm(d_1.d_2\cdots d_n)_{\beta_1} \times \beta_2^E\,. \tag{11}$$

In the above, $\pm$ is the sign of the number. The factor $(d_1.d_2\cdots d_n)_{\beta_1}$ is known as the significand. The digits after the decimal point, $d_2\cdots d_n$, are called the fraction or mantissa. β_1 is the base for the fraction; β_2 is the base for the exponent. E is the exponent. The floating-point form of a real number is called *normal* if $d_1 \neq 0$; note $0 \leq d_i < \beta_1$. In normal form one also has the relation that $(d_1.d_2\cdots d_n)_{\beta_1} < \beta_1$

In order to store a real number expressed in normal form, the computer needs to allocate space for the sign, fraction, and exponent – there is no need to store d_1 since in the binary system it is always equal to 1. In most computer systems real numbers are stored using 32-bits (*single precision*) or 64-bits (*double precision*). For both integers and real numbers, the precise word-length is dependent on the computer architecture and can vary widely from system to system.[1]

2.3 IEEE standard

MATLAB uses the IEEE standard for storing floating point numbers. Its default is to use double precision numbers. However, we will start by discussing single precision numbers as it requires less writing. For single precision IEEE numbers (32 bits), the first bit is the sign bit, the next 8 bits are for the *biased* exponent (a shifted form of the exponent), and the remaining 23 bits are for the fraction (mantissa). Both the exponent and the fraction are stored in base 2 so that $\beta_1 = \beta_2 = 2$. A single precision number, V, "looks like":

$$V = \underbrace{\sqcup}_{\text{sign bit }(s)}\quad \underbrace{\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup}_{\text{biased exponent bits }(e)}\quad \underbrace{\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup}_{\text{mantissa bits }(f)} \tag{12}$$

[1]When buying computers and/or operating systems, they are often designated as 32-bit or 64-bit. This designation refers to the word length used to index locations in memory. Thus a 64-bit computer/OS can utilize a lot more memory than a 32-bit computer/OS. But this is independent of the sizes used to store variables in a program.

By definition: If all the biased exponent bits are set, i.e. $e = 255$, and f is nonzero then $V = \text{NaN}$ (Not a Number). If all the biased exponent bits are set, i.e. $e = 255$, $f = 0$ and $s = 1$, then $V = -\infty$. If all the biased exponent bits are set, i.e. $e = 255$, $f = 0$ and $s = 0$, then $V = +\infty$. If $0 < e < 255$, then

$$V = (-1)^s \times 2^{e-127} \times (1.f)_2 \,. \tag{13}$$

The "$(1.f)_2$" represents the binary number created by appending f to an implicit "1.". The actual exponent E is not stored. Rather, $e = E + 127$ is stored so that the exponent can have positive as well as negative values without the need for another sign bit; 127 is referred to as the *bias*. Beyond normal numbers there is also the special case of subnormal numbers. A subnormal number occurs when $e = 0$. In this situation, the significand is assumed to be of the form $(0.f)_2$. On some systems, numbers of this form cause *underflow errors*. The most important subnormal number is the case where $f = 0$; in which case, $V = \pm 0$ depending on the sign bit. There are similar rules for double precision numbers.

It is not critical to memorize exactly how IEEE numbers are stored. What really matters is understanding that there are limits on what can be stored and computed. For example, there is a minimum(non-zero) and maximum value that can be stored. The maximum value of an IEEE 32-bit number is

$$\underbrace{0}_{s}\ \underbrace{11111110}_{e}\ \underbrace{11111111111111111111111}_{f} = (-1)^0 \times 2^{254-127} \times (2-2^{-23}) \approx 3.40 \times 10^{38} \tag{14}$$

The term $2 - 2^{-23}$ comes from exploiting the relation for the geometric sum:

$$1 + a^1 + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a} \,; \tag{15}$$

i.e.,

$$(1.11111111111111111111111)_2 = (1/2)^0 + (1/2)^1 + \cdots + (1/2)^{23} = \frac{1 - (1/2)^{23+1}}{1 - (1/2)} = 2 - 2^{-23} \tag{16}$$

The minimum(non-zero) normal number representable in IEEE single-precision is

$$\underbrace{0}_{s}\ \underbrace{00000001}_{e}\ \underbrace{00000000000000000000000}_{f} = (-1)^0 \times 2^{1-127} \times (1) \approx 1.2 * 10^{-38} \tag{17}$$

MATLAB's default for storing numbers is IEEE double precision floating-point format in which there is one sign bit, 11 (biased) exponent bits, and 52 mantissa bits; the bias is 1023. The biased exponent e is required to be in the range $0 < e < 2047$ for normal numbers:

$$V = (-1)^s \times 2^{e-1023} \times (1.f)_2 \,. \tag{18}$$

Rules for NaN, $\pm\infty$, etc. follow as above with the critical value of e being 2047 instead of 255.

3 Finite precision of numbers

The fact that numbers are stored using a finite number of bits for the exponent and the mantissa has important implications on the accuracy of numerical computations. It is quite possible to have very well formulated mathematical expressions that simply do not compute as one would expect.

Example 3 Let us consider adding a number to unity; i.e., $1+\delta$. Because of the way the numbers are stored, we could for instance obtain the seemingly odd result that $1+\delta = 1$ for $\delta > 0$. To see how this could occur we need to think about how the simple operation of addition could be performed inside a computer. Suppose we start with the number 1; in IEEE double precision, it has an exponent $E = 0$ and a biased exponent $e = 1023 = (01111111111)_2$ and mantissa $f = 0$. If we are to try and make as small a change as possible to 1 we would take the mantissa and add a "1" to the very last bit; i.e. make $f = 2^{-52}$ (recall there are 52 bits in the double precision mantissa). This represents adding $\approx 2.2 \times 10^{-16}$ to 1. If we try to add a number smaller than 2^{-52} to 1 there are not enough mantissa bits to be able to represent the result. The computer simply throws away these extra bits and the information is lost. It is easy to check this fact in MATLAB with a simple program:

```
for ii = -50:-1:-54
  if  1 == (1 + 2^ii)
    fprintf('1 + 2^%d equals 1\n',ii);
  else
    fprintf('1 + 2^%d does not equal 1\n',ii);
  end
end
```

The output of which is

```
1 + 2^-50 does not equal 1
1 + 2^-51 does not equal 1
1 + 2^-52 does not equal 1
1 + 2^-53 equals 1
1 + 2^-54 equals 1
```

The number 2^{-52} is known as *machine epsilon*; it is simply the smallest number one can add to unity and still be able to detect it. In MATLAB *machine epsilon* is the pre-defined variable `eps`. When represented as a base 10 number, *machine epsilon*$\approx 2.2 \times 10^{-16}$ and this tells us that in general when one adds two numbers (in IEEE double precision) there are 16 (base 10) digits of accuracy in the computation; i.e. `(a + a*b) == a`, if `b < eps`.[2]

[2]To have MATLAB display more digits of a number, in the command window, you should type `format long e` at the command prompt. For other printing format options type `help format`.

Example 4 Consider a hypothetical computer system constructed upon a base 10 number system that can store 3 digits in the significand (leading digit plus mantissa). Now consider adding $x = 2 = 10^0 \times 2.00$ to $y = 0.001 = 10^{-3} \times 1.00$. In order for the computer to perform this operation it needs to have both numbers utilizing the same exponent. This makes it possible to directly add the significands. The computation will look like

$$\begin{array}{rccccccl} & 2 & . & 0 & 0 & | & & \times 10^0 \\ + & 0 & . & 0 & 0 & | & 1 & \times 10^0 \\ \hline & 2 & . & 0 & 0 & | & 1 & \times 10^0 \end{array} \tag{19}$$

Note that to get the exponent of y to match that of x, the 1 in necessarily shifted beyond the 3 digits that the our hypothetical computer stores. The result will be that our computer will tell us that $x + y$ is 2.00.

Example 5 As a last example consider the innocuous looking request to MATLAB: `>> 1.0 − 0.8 − 0.2`. Type this into MATLAB and one finds that the result is not zero! However `>> 1.0 − 0.2 − 0.8` is zero. This occurs even though MATLAB is using 11 biased exponent bits and 52 mantissa bits. The reason becomes apparent if one carefully looks at how these numbers are stored in IEEE format. The number 1 looks like:

$$\underbrace{0}_{1\ \text{bit}}\ \underbrace{01111111111}_{11\ \text{bits}}\ \underbrace{0000000000000000000000000000000000000000000000000000}_{52\ \text{bits}} \tag{20}$$

The number 0.8 appears as

$$\underbrace{0}_{1\ \text{bit}}\ \underbrace{01111111110}_{11\ \text{bits}}\ \underbrace{1001100110011001100110011001100110011001100110011001}_{52\ \text{bits}} \tag{21}$$

and the number 0.2 appears as

$$\underbrace{0}_{1\ \text{bit}}\ \underbrace{01111111100}_{11\ \text{bits}}\ \underbrace{1001100110011001100110011001100110011001100110011001}_{52\ \text{bits}} \tag{22}$$

The first thing to note is that in base 2 these decimals repeat infinitely. They can not be stored exactly – similar to how 1/3 can not be stored exactly in a finite number of base 10 digits. When the numbers are shifted to effect the computations, bits get lost and result in errors whereby `1.0−0.8−0.2` is not equal to `1.0−0.2−0.8`. The difference however can be determined by noting that the errors will be associated with incorrect trailing bits in the IEEE format. In this particular case, with a bit of effort, one can show that the error occurs in the second to last bit – the one with value $2^{-3} \times 2^{-51}$ (-3 is the exponent for 0.2). In fact:

```
>> 1.0−0.8−0.2 + 2^−54
ans =
     0
```

One would never try to implement this type of brute force correction scheme to improve precision as it is quite computation dependent. Instead in programs that depend upon floating point computations, care is taken when comparing numbers to ensure that one is not asking for precision that is not computable by the computer.

Remark: To convert a base 10 number to IEEE format one first computes the true exponent as $E = \lfloor \log_2(|x|) \rfloor$. The IEEE exponent is then the binary representation of E shifted by the bias. The mantissa can be found by reducing to normal form and subtracting the leading 1: $f = (|x|/2^E) - 1$. Finally the mantissa can be converted to binary by successively subtracting powers of 2^n for $n \in \{-1, -2, \cdots, -52\}$ – a process similar to what was done in Example 2 (but without the need for determining the digit value since it can only be 1 or 0 in a binary representation).