Digital Representation Of Data

Scientific Computing
Fall, 2018
Paul Gribble

Here we review how data are stored in a digital format on computers.

1 Binary

Information on a digital computer is stored in a binary format. Binary format represents information using a series of 0s and 1s. If there are n digits of a binary code, one can represent 2^n bits of information.

So for example the binary number denoted by:

```
0001
```

represents the number 1. The convention here is called little-endian because the least significant value is on the right, and as one reads right to left, the value of each binary digit doubles. So for example the number 2 would be represented as:

```
0010
```

This is a 4-bit code since there are 4 binary digits. The full list of all values that can be represented using a 4-bit code are shown in Table 1.

So with a 4-bit binary code one can represent $2^4 = 16$ different values (0-15). Each additional bit doubles the number of values one can represent. So a 5-bit code enables us to represent 32 distinct values, a 6-bit code 64, a 7-bit code 128 and an 8-bit code 256 values (0-255).

Another piece of terminology: a given sequence of binary digits that forms the natural unit of data for a given processor (CPU) is called a word.

Have a look at the ASCII table. The standard ASCII table represents 128 different characters and the extended ASCII codes enable another 128 for a total of 256 characters. How many binary bits are used for each?

2 Hexadecimal

You will also see in the ASCII table that it gives the decimal representation of each character but also the Hexadecimal and Octal representations. The hexadecimal system is a base-16 code and the octal system is a

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Table 1: Binary and decimal values for a 4-bit code.

base-8 code. Hex values for a single hexadecimal digit can range over:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

If we use a 2-digit hex code we can represent $16 * 16 = 256$ distinct values. In computer science, engineering and programming, a common practice is to represent successive 4-bit binary sequences using single-digit hex codes.

Table 2 shows 4-bit values of Binary, Decimal and Hexadecimal.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Table 2: Binary, Decimal and Hexadecimal values for a 4-bit code.

If we have 8-bit binary codes we would use successive hex digits to represent each 4-bit word of the 8-bit byte (another piece of lingo). Table 3 shows how this would look for some 8-bit values in binary, decimal and hexadecimal.

Binary	Decimal	Hexadecimal
0000 0000	0	00
0000 0001	1	01
0000 0010	2	02
...	...	...
1111 1101	253	FD
1111 1110	254	FE
1111 1111	255	FF

Table 3: Binary, Decimal and Hexadecimal values for an 8-bit (1-byte) code.

The left chunk of 4-bit binary digits (the left word) is represented in hex as a single hex digit (0-F) and the next chunk of 4-bit binary digits (the right word) is represented as another single hex digit (0-F).

Hex is typically used to represent bytes (8-bits long) because it is a more compact notation than using 8 binary digits (hex uses just 2 hex digits).

3 Floating point values

The material above talks about the decimal representation of bytes in terms of integer values (e.g. 0-255). Frequently however in science we want the ability to represent real numbers on a continuous scale, for example 3.14159, or 5.5, or 0.123, etc. For this, the convention is to use floating point representations of numbers.

The idea behind the floating point representation is that it allows us to represent an approximation of a real number in a way that allows for a large number of possible values. Floating point numbers are represented to a fixed number of *significant digits* (called a *significand*) and then this is scaled using a *base* raised to an *exponent*:

$$s \times b^e \tag{1}$$

This is related to something you may have come across in high-school science, namely scientific notation. In scientific notation, the base is 10 and so a real number like 123.4 is represented as 1.234×10^2.

In computers there are different conventions for different CPUs but there are standards, like the IEEE 754 floating-point standard. As an example, a so-called single-precision floating point format is represented in binary (using a base of 2) using 32 bits (4 bytes) and a /double precision/ floating point number is represented using 64 bits (8 bytes). In C you can find out how many bytes are used for various types using the `sizeof()` function:

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("a single precision float uses %ld bytes\n", sizeof(float));
  printf("a double precision float uses %ld bytes\n", sizeof(double));
  return 0;
}
```

On my macbook pro laptop this results in this output:

```
a single precision float uses 4 bytes
a double precision float uses 8 bytes
```

According to the IEEE 754 standard, a single precision 32-bit binary floating point representation is composed of a *1-bit sign bit* (signifying whether the number is positive or negative), an *8-bit exponent* and a *23-bit significand*. See the various wikipedia pages for full details.

There is a key phrase in the description of floating point values above, which is that floating point representation allows us to store an *approximation* of a real number. If we attempt to represent a number that has more significant digits than can be store in a 32-bit floating point value, then we have to approximate that real number, typically by rounding off the digits that cannot fit in the 32 bits. This introduces rounding error.

Now with 32 bits, or even 64-bits in the case of double precision floating point values, rounding error is likely to be relatively small. However it's not zero, and depending on what your program is doing with these values, the rounding errors can accumulate (for example if you're simulating a dynamical system over thousands of time steps, and at each time step there is a small rounding error).

We don't need a fancy simulation however to see the results of floating point rounding error. Open up your favourite programming language (MATLAB, Python, R, C, etc) and type the following (adjust the syntax as needed for your language of choice):

```
(0.1 + 0.2) == 0.3
```

What do you get? In MATLAB I get:

```
>> (0.1 + 0.2) == 0.3

ans =

     0
```

In MATLAB, `0` is synonymous with the logical value `FALSE`. What's going on here? What's happening is that these decimal numbers, 0.1, 0.2 and 0.3 are being represented by the computer in a binary floating-point format, that is, using a base 2 representation. The issue is that in base 2, the decimal number 0.1 cannot be represented precisely, no matter how many bits you use. Plug in the decimal number 0.1 into an online binary/decimal/hexadecimal converter (such as here) and you will see that the binary representation of 0.1 is an infinitely repeating sequence:

```
0.0001100110011001100110011001100... (base 2)
```

This shouldn't be an unfamiliar situation, if we remember that there are also real numbers that cannot be represented precisely in decimal format, either, because they involve an infintely repeating sequence. For example the real number $\frac{1}{3}$ when represented in decimal is:

```
0.3333333333... (base 10)
```

If we try to represent $\frac{1}{3}$ using n decimal digits then we have to chop off the digits to the right that we cannot include, thereby rounding the number. We lose some amount of precision that depends on how many significant digits we retain in our representation.

So the same is true in binary. There are some real numbers that cannot be represented precisely in binary floating-point format.

See here for some examples of significant adverse events (i.e. disasters) cause by numerical errors.

Rounding can be used to your advantage, if you're in the business of stealing from people (see salami slicing). In the awesomely kitchy 1980s movie Superman III, Richard Pryor's character plays a "bumbling computer genius" who embezzles a ton of money by stealing a large number of fractions of cents (which in the movie are said to be lost anyway due to rounding) from his company's payroll (YouTube clip here).

There is a comprehensive theoretical summary of these issues here: What Every Computer Scientist Should Know About Floating-Point Arithmetic.

Also see these webpages from the MathWorks online documentation about how MATLAB represents floating-point numbers:

Floating-Point Numbers

and this section on avoiding common problems with Floating-Point Arithmetic:

Avoiding Common Problems with Floating-Point Arithmetic

Here is a fantastic blog post that takes you through how floating-point numbers are represented:

Exposing Floating Point

3.1 Integer Overflow

Just in case you thought that floating point values are the only source of problems, representing integer values also comes with the problem of *integer overflow*. This is when one attempts to represent an integer that is larger than possible given the number of bits available.

So for example if we were representing positive integers using only 16 bits, we would only be able to store $2^{16} = 65536$ distinct values. So if the first value is 0 then we are able to store positive integers up to 65535. If we attempt to add the value 1 to a variable that uses 16 bits and is currently storing the value 65535, the variable will "overflow", probably back to zero, in this case.

Here is a not-well-enough-known recent case of integer overflow error affecting Boeing's new 787 "Dreamliner" aircraft:

Reboot Your Dreamliner Every 248 Days To Avoid Integer Overflow

4 ASCII

ASCII stands for *American Standard Code for Information Interchange*. ASCII codes delineate how text is represented in digital format for computers (as well as other communications equipment).

ASCII uses a 7-bit binary code to represent 128 specific characters of text. The first 32 codes (decimal 0 through 31) are non-printable codes like `TAB`, `BEL` (play a bell sound), `CR` (carriage return), etc. Decimal codes 32 through 47 are more typical text symbols like `#` and `&`. Decimal codes 48 through 57 are the numbers `0` through `9`. Decimal codes 65 through 90 are capital letters `A` through `Z`, and codes 97 through 122 are lowercase letters `a` through `z`. Table 4 shows codes in decimal, hexadecimal and octal (base-8) for the numbers 0 through 9. Table 5 shows codes for uppercase and lowercase letters.

For a full description of the 7-bit ascii codes in their entirety, including the *extended ASCII codes* (where you will find things like ö and é), see this webpage:

http://www.asciitable.com (ASCII Table and Extended ASCII Codes).

Dec	Hex	Oct	Chr
48	30	060	0
49	31	061	1
50	32	062	2
51	33	063	3
52	34	064	4
53	35	065	5
54	36	066	6
55	37	067	7
56	38	070	8
57	39	071	9

Table 4: 7-bit ASCII codes for the numbers 0 through 9.

In MATLAB, all individual text characters (variable type `char`) are represented, under the hood, as decimal ASCII values. Have a look at this code, in which we ask for the numeric value of individual characters. You can see that the result corresponds to their decimal ASCII values in Table 5.

```
>> double('a')

ans =

    97

>> double('b')

ans =

    98

>> double('z')

ans =

   122
```

You can get the character value of an ASCII code in MATLAB using the `char()` function:

```
>> char(65)

ans =

A
```

You can use your knowledge of ASCII codes to do tricky things in MATLAB, like convert to and from uppercase and lowercase, given your knowledge that the difference (in decimal) between ASCII `A` and ASCII `a` is 32 (see Table 5).

```
>> char('A' + 32)

ans =

a
```

Dec	Hex	Oct	Chr	Dec	Hex	Oct	Chr
65	41	101	A	97	61	141	a
66	42	102	B	98	62	142	b
67	43	103	C	99	63	143	c
68	44	104	D	100	64	144	d
69	45	105	E	101	65	145	e
70	46	106	F	102	66	146	f
71	47	107	G	103	67	147	g
72	48	110	H	104	68	150	h
73	49	111	I	105	69	151	i
74	4A	112	J	106	6A	152	j
75	4B	113	K	107	6B	153	k
76	4C	114	L	108	6C	154	l
77	4D	115	M	109	6D	155	m
78	4E	116	N	110	6E	156	n
79	4F	117	O	111	6F	157	o
80	50	120	P	112	70	160	p
81	51	121	Q	113	71	161	q
82	52	122	R	114	72	162	r
83	53	123	S	115	73	163	s
84	54	124	T	116	74	164	t
85	55	125	U	117	75	165	u
86	56	126	V	118	76	166	v
87	57	127	W	119	77	167	w
88	58	130	X	120	78	170	x
89	59	131	Y	121	79	171	y
90	5A	132	Z	122	7A	172	z

Table 5: 7-bit ASCII codes for uppercase and lowercase letters.

```
>> char('a' - 32)

ans =

A
```