PHYC 2050 Class Notes

Thomas J. Duck

Department of Physics and Atmospheric Science
Dalhousie University

tom.duck@dal.ca

February 17, 2010

CHAPTER

THREE

NUMERICAL PRECISION

The limits of machine precision should be considered when performing calculations. We have already seen examples of numbers how numbers are rounded off to a certain precision. For example numpy's representation of π isn't infinitely long:

```
>>> numpy.pi
3.1415926535897931
```

More suprisingly, evaluating 0.1 at the python prompt returns the following result:

```
>>> 0.1
0.10000000000000001
```

The reasons for and implications of limited machine precision are explored in this chapter.

3.1 Representation of Numbers on a Computer

The fundamental unit of information on a computer is the *bit*, which may have *binary* values 0 or 1, corresponding to the ON/OFF states in a microchip transistor. All numbers, letters, and machine instructions on a computer are encoded in binary.

Consider the representation of numbers in 3 bits. The possible permutations of 0 and 1 in three bits are

000 001 010 011 100 101 110 and 111

and so we can represent only eight different numbers in three bits. We have said nothing yet of the interpretation of each permutation. Integers and floating-point numbers are encoded differently, and we will explore details shortly.

More generally, a series of N bits has 2^N permutations of 0s and 1s. Put another way, 2^N numbers can be represented in N bits. Since N is necessarily finite (no computer has infinite memory), it follows that not all numbers can be represented exactly.

Computers are hard-wired for arithmetic with numbers encoded in a specific number of bytes (8 bit units). Typically this means 4 bytes (32 bits) or 8 bytes (64 bits), which yields 4.3 billion and 18 quintillion unique numbers, respectively.

Note: Machines with 32-bit CPU registers often support hard-wired 64-bit mathematics via on-CPU or separate math co-processors. Also, other sizes that are not supported in hardware can be emulated in software.

3.1.1 Numbers in any Base

We are all familiar with the decimal number system, which uses the digits 0-9 and so is said to have base-10. Maybe you also know the binary number system, which uses the two digits 0 and 1 and so is said to be base-2. Less familar may be the octal (base-8) and hexadecimal (base-16) number systems.

It is easy to write numbers in any base once we fundamentally understand how numbers are written in base-10. The decimal system has digits written in columns (see Figure 3.1): the ones, tens, hundreds, etc columns to the left of the decimal point, and the tenths, hundreds, thousandths columns to the right.

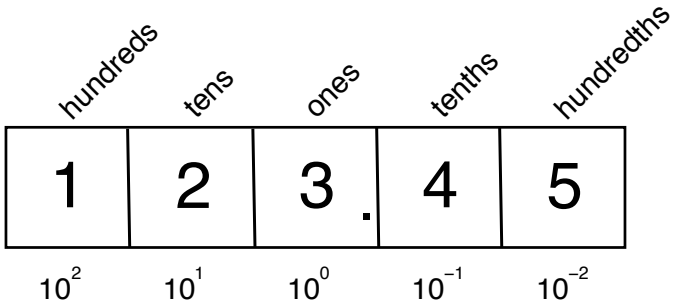

Figure 3.1: The decimal number 123.45 with column names on the top, and as powers of ten on the bottom.

For the number 123.45 we say there is one group of a hundred, two groups of ten, three groups of one, etc. Notice that the columns can be marked as sequential powers of ten.

The base-2 (binary) number system is analogous (see Figure 3.2). Instead of powers of ten for the columns, we use powers of two. Whole-number columns are to the left of the radix point (formerly called the decimal point) and fractions are to the right. Because we only have the digits 0 and 1 to work with, there can only be zero or one group for each column. Converting to decimal is as simple as summing up the powers of two for each column containing a one.

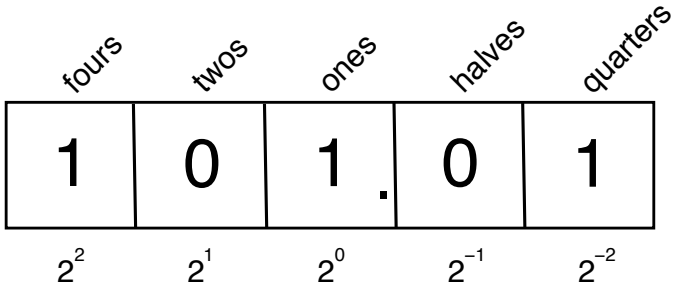

Figure 3.2: The binary number 101.01 with column names on the top, and as powers of two on the bottom. The binary representation converts to decimal as $4 + 1 + 0.25 = 5.25$.

Base-8 (octal, digits 0-7) and base-16 (hexadecimal, digits 0-9 and A-F) follow in a similar fashion. With this basic understanding of how the different number systems work, we can proceed to a more formal description of how integers and floating-point numbers are represented in computer memory.

3.1.2 Integers

Following the example in the previous section, the formula for conversion of base-2 integers to base-10 is

$$(d_{M-1}d_{M-2}\ldots d_1d_0)_2 = \sum_{i=0}^{M-1} d_i 2^i$$

where M is the number of digits and the subscript 2 is used to indicate the base. The digits are constrained such that $d_i \in [0, 1]$. For example, 1101_2 converts to decimal using the formula above as follows:

$$\begin{aligned} 1101_2 &= (1)\cdot 2^3 + (1)\cdot 2^2 + (0)\cdot 2^1 + (1)\cdot 2^0 \\ &= 13 \end{aligned}$$

Standard integers are stored in 4 bytes (32 bits). There are $2^{32} = 4294967296$ (4.3 billion) permutations of 0s and 1s in 32 bits. If one bit is used to encode the sign, then the range for 32-bit integers is -2^{31} to $2^{31} - 1$; i.e., -2147483648 to 2147483647. This can be demonstrated in the python interpreter:

```
>>> numpy.int32(2**31-1)                # Should be OK
2147483647

>>> numpy.int32(2**31)                  # Should be out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: long int too large to convert to int

>>> numpy.int32(-2**31)                 # Should be OK
-2147483648

>>> numpy.int32(-2**31 - 1)             # Should be out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: long int too large to convert to int
```

Note: The actual encoding of integers in computer memory is done in something called "Two's Complement" in order to facilitate electronic computations. That is not important here.

The general conversion formula for an integer with base B and M digits is

$$(d_{M-1}d_{M-2}\ldots d_1d_0)_B = \sum_{i=0}^{M-1} d_i B^i$$

where digits $d_i \in [0, 1, \ldots, B-2, B-1]$.

The following table provides the first sixteen integers as written in decimal, binary, actal and hexadecimal. Special prefixes are used for octal (0) and hexidecimal (0x) instead of subscripts.

Decimal	Binary	Octal	Hexadecimal
0	0_2	00	0x0
1	1_2	01	0x1
2	10_2	02	0x2
3	11_2	03	0x3
4	100_2	04	0x4
5	101_2	05	0x5
6	110_2	06	0x6
7	111_2	07	0x7
8	1000_2	010	0x8
9	1001_2	011	0x9
10	1010_2	012	0xA
11	1011_2	013	0xB
12	1100_2	014	0xC
13	1101_2	015	0xD
14	1110_2	016	0xE
15	1111_2	017	0xF

Octal and hexadecimal are useful because one digit of each represents 3 and 4 bits of binary, respectively.

e.g.: Consider 111101101_2:

```
Binary:       111 101 101       (in groups of 3 bits)
Octal:      0   7   5   5

Binary:    0001 1110 1101       (in groups of 4 bits)
Hex:    0x    1    E    D
```

You can use the python `int()`, `oct()` and `hex()` built-in functions to convert integers between different bases. Note in particular that the `int()` function accepts data:*base* as a parameter.

3.1.3 Floating Point Numbers

How can we write numbers with a radix point? We could, for example, extend the framework for integers, and choose the radix point to be fixed to the right of the first digit. That would allow tiny fractions (although not small enough), but would not allow us to represent very large numbers. If we were to choose the radix point to be fixed to the left of the last digit then that would allow larger numbers (still not large enough) but would no longer allow us to write fractions like 0.25. We could choose to have the radix point somewhere in between and then have the worst of both worlds. Indeed, fixed-point representation is a poor way to represent numbers with a radix point for most applications (accounting software being a notable exception).

Numbers with a radix point are best written in scientific notation; e.g., 1.2345×10^2. In this case the radix point floats so that we have consistent precision in any number we choose to write. Hence, they are also called floating-point numbers.

To see how floating-point numbers are represented in binary we will again revisit the more familar decimal system. Decimal numbers are written in scientific notation using the form

$$\pm d_0.d_1d_2d_3\ldots d_{p-1} \times 10^n$$

where p is the precision (number of significant digits), n is the integer exponent, and the digits d_i are constrained such that

$$d_i = \begin{cases} \in [1,\ldots,9] & \text{if } i = 0, \\ \in [0,\ldots,9] & \text{otherwise.} \end{cases}$$

The *significand* $d_0.d_1d_2d_3\ldots d_{p-1}$ can be written in sum form as:

$$\pm d_0.d_1d_2d_3\ldots d_{p-1} = \pm\sum_{i=0}^{p-1} d_i 10^{-i}$$

More generally, for arbitrary base B we can write floating-point numbers using the form

$$\pm(d_0.d_1d_2d_3\ldots d_{p-1})_B \times B^n$$

where

$$d_i = \begin{cases} \in [1,\ldots,B-1] & \text{if } i = 0 \\ \in [0,\ldots,B-1] & \text{otherwise} \end{cases}$$

and where the significand is converted to decimal using

$$(d_0.d_1d_2d_3\ldots d_{p-1})_B = \sum_{i=0}^{p-1} d_i B^{-i}$$

Binary floating point numbers can therefore be converted with

$$\pm(1.d_1d_2d_3\ldots d_{p-1})_2 \times 2^n$$

Notice that the first digit can only be 1 if the proper scientific notation as described by the general formalism is to be used. Thus, binary floating-point numbers can be written as

$$\pm(1+f)_2 \times 2^n$$

where $d_i \in [0,1]$, and f is the fraction with $0 \leq f < 1$. The fraction with $m = p-1$ digits to the right of the radix point is converted to decimal using

$$f = (0.d_1d_2\ldots d_{m-1}d_m)_2 = \sum_{i=1}^{m} d_i 2^{-i}$$

Thus, the sign, binary exponent, and binary fraction digits can be used to minimally store a floating-point number in computer memory.

Example

What is the spacing of binary fraction values if three digits are used?

The sequence of available binary fraction values obtained from the above formula is

$$0.000_2, 0.001_2, 0.010_2, 0.011_2, \ldots = 0.000, 0.125, 0.25, 0.375, \ldots$$

The fraction values are uniformly spaced by 0.125. Uniform spacing is a general feature of the floating-point number system. The spacing of nearest-neighbour fraction values is 2^{-m}, where m is the number of digits to the right of the radix point.

The IEEE 754 Standard

The IEEE 754 Standard specifies ways to store binary floats in memory. "Double precision" floating-point numbers are stored in eight bytes (64 bits), so $2^{64} = 1.8 \times 10^{19}$ (18 quintillion) different double-precision floats are available.

The bit layout for double precision is as follows:

Figure 3.3: Bit layout for IEEE 754 double-precision floating-point numbers.

The exponent n is a signed integer that has $2^{11} = 2048$ permutations in the 11 bits available. It ranges from -1022 to 1023 with the 2 remaining values reserved for special cases (zero, Inf, NaN, and subnormal values). The fraction portion stores only the binary digits to the right of the radix point.

Note: The exponent encoding is actually "biased" to facilitate computations. This has no practical implications for our work.

Examples

e.g. What is the largest positive double-precision float?

From the previous example, the spacing of fraction values is 2^{-52}. The largest fraction is therefore $f = 1 - 2^{-52} \approx 1$. The largest exponent is $n = 1023$. Therefore, the largest double-precision float is

$(1 + 1) \times 2^{1023} = 2^{1024} \approx 1.798 \times 10^{308}$

Numbers greater than this produce "overflow errors":

```
>>> 1.797e308
1.797e+308
>>> 1.799e308
inf
```

e.g., What is the smallest positive double-precision float?

The smallest fraction is $f = 0$. The smallest exponent is $n = -1022$. Therefore, the smallest (normalized) positive double-precision float is

$(1 + 0) \times 2^{-1022} \approx 2.225 \times 10^{-308}$

There are, however, a set of "denormalized" numbers with significand $0 + f$ that have reduced precision. In this case, the minimum value is

$$(0 + 2^{-52}) \times 2^{-1022} = 2^{-1074} \approx 4.9 \times 10^{-324}$$

Numbers as low as half this get rounded up, while those smaller than half this are "underflow errors" and may be rounded to zero

```
>>> 2.5e-324               # Should get rounded up
4.9406564584124654e-324

>>> 2.4e-324               # Underflow
0.0
```

e.g. Can 1.25 be represented as a double-precision float?

Yes.

Choose exponent $n = 0$ since $2^0 \leq x < 2^1$.

$\rightarrow 1.25 = (1 + f) \times 2^0$

$\rightarrow f = 0.25 = 0.01_2$

Notice that f is between 0 and 1; choosing the exponent n in any other way would cause this requirement to fail. Note also that f has two binary digits to the right of the radix point.

IEEE 754 double-precision floats require that $-1022 \leq n < 1023$ and that f has no more than 52 binary digits right of the radix point. Since both n and f can be written within the constraints of the numbering system, 1.25 is represented exactly.

e.g. Can 0.1 be represented as a double-precision float?

No.

Choose $n = -4$ since $2^{-4} \leq x < 2^{-3}$.

$\rightarrow 0.1 = (1 + f) \times 2^{-4}$

$\rightarrow f = 0.6 = 0.\overline{1001}_2$

You can convince yourself of this using the python interpreter by adding successive values of $2^{-i} = 0.5 * *i$ and zeroing out the terms that put the sum over 0.6. The first twelve terms are

```
(1)*0.5**1 + (0)*0.5**2 + (0)*0.5**3 + (1)*0.5**4 + (1)*0.5**5 + (0)*0.5**6 + (0)*0.5**7
+ (1)*0.5**8 + (1)*0.5**9 + (0)*0.5**10 + (0)*0.5**11 + (1)*0.5**12
```

which reveals the repeating pattern. This approach is consistent with the general formulas given above.

Since f cannot be written in 52 binary digits, 0.1 cannot be represented exactly and is rounded off. Hence the following result when 0.1 is evaluated in the python interpreter:

```
>>> 0.1
0.10000000000000001
```

e.g. What is the spacing between adjacent double-precision numbers for exponent $n = 0$? What about $n = 50$?

The fraction values are spaced by 2^{-52}. For $n = 0$, the spacing between floating-point numbers is $2^{-52} \times 2^0 \approx 2.2 \times 10^{-16}$. For $n = 50$, the spacing between floating-point numbers is $2^{-52} \times 2^{50} = 0.25$. Therefore, large numbers are more widely spaced.

3.2 Machine Epsilon and Roundoff Error

Floating point numbers that cannot be encoded exactly are rounded off. The maximum relative error when rounding off to the nearest float is called "machine epsilon".

The smallest floating point number for exponent n is

$(1 + 0) \times 2^n = 2^n$.

The spacing between adjacent values for m fraction digits and exponent n is

$2^{-m} \times 2^n = 2^{-m+n}$.

The maximum absolute error when rounding off is half this; i.e.,

$\frac{1}{2} \times 2^{-m+n} = 2^{-m+n-1}$.

The relative error is therefore

$$\epsilon = \frac{2^{-m+n-1}}{2^n}$$

$$\rightarrow \epsilon = 2^{-m-1}$$

For IEEE 754 floats ($m = 52$), $\epsilon = 2^{-53} \approx 1.11 \times 10^{-16}$.

Another way to interpret machine epsilon is that it is the smallest number such that $1 + \epsilon > 1$. Why? Since the maximum absolute error for numbers near 1 is $1 \times \epsilon = \epsilon$, adding ϵ to 1 should force roundoff to the next greater nearest neighbour:

```
>>> 2.**(-53)            # Can't be represented exactly; gets rounded down
1.1102230246251565e-16

>>> 1. + 1.1102230246251565e-16
1.0

>>> 1. + 1.1102230246251567e-16  # Round it up a bit
1.0000000000000002
```

Machine epsilon has consequences for our calculations. For example, in mathematics, addition is associative. i.e.:

$$(a + b) + c = a + (b + c)$$

Suppose $a = 1$, $b = \epsilon/2$ and $c = \epsilon/2$, and evaluated the two sides as a computer would:

$LHS = (1 + \epsilon/2) + \epsilon/2 = 1 + \epsilon/2 = 1$

$RHS = 1 + (\epsilon/2 + \epsilon/2) = 1 + \epsilon$

$\rightarrow LHS \neq RHS$

Apparently addition is not necessarily associative for floating point mathematics on a computer. The order of operations sometimes matters.

We will need to be very careful about subtracting nearly equal numbers. For example:

```
>>> eps = 1.1102230246251567e-16      # Rounded up
>>> (1+eps) - 1           # Should return eps; returns twice that
2.2204460492503131e-16
```

This shows that there is a loss of precision when subtracting nearly equal numbers – such as is the case for the finite difference estimate of derivatives, which was discussed in the previous chapter. Where possible, this problem should be avoided by reorganizing the calculations.

As one final example of what can go wrong, try taking the cosine of integer multiples of 2π. The result should always be 1.0, but in the second case below it goes spectacularly wrong:

```
>>> numpy.cos( 2*numpy.pi )
1.0


>>> numpy.cos( 4e16 * 2*numpy.pi )
-0.99789442632593861
```

The problem is that the absolute error in the argument to the cosine function is $\epsilon \times (4 \times 10^{16} \times 2\pi) \approx 28$, which is much larger even than the 2π radians for a complete cycle. For this reason, arguments to the trigonometric functions are usually kept within a few π of zero.