Computer numbers

October 4, 2014

References:

- Gentle, Computational Statistics, Chapter 2.
- http://www.lahey.com/float.htm
- And for more gory detail, see Monahan, Chapter 2.

A quick note that, as we've already seen, R's version of scientific notation is *XeY*, which means $X \cdot 10^Y$.

A second note is that the concepts developed here apply outside of R, but we'll illustrate the principles of computer numbers using R.

1 Basic representations

Everything in computer memory or on disk is stored in terms of bits. A *bit* is essentially a switch than can be either on or off. Thus everything is encoded as numbers in base 2, i.e., 0s and 1s. 8 bits make up a *byte*. For information stored as plain text (ASCII), each byte is used to encode a single character (actually 7 bits are used, hence there are $2^7 = 128$ ASCII characters). One way to represent a byte is to write it in hexadecimal, rather than as 8 0/1 bits. Since there are $2^8 = 256$ possible values in a byte, we can represent it more compactly as 2 base-16 numbers, such as "3e" or "a0" or "ba". A file format is nothing more than a way of interpreting the bytes in a file.

We can think about how we'd store an integer in terms of bytes. With two bytes, we could encode any value from $0, \ldots, 2^{16} - 1 = 65535$. This is an unsigned integer representation. To store negative numbers as well, we can use one bit for the sign, giving us the ability to encode -32767 - 32767 ($\pm 2^{15} - 1$). Note that in general, rather than be stored simply as the sign and then a number in base 2, integers are actually stored in a different binary encoding to facilitate arithmetic. Finally note that the set of computer integers is not closed under arithmetic, with R reporting an overflow (i.e., a result that is too large to be stored as an integer):

```r
a <- as.integer(3423333)  # 3423333L
a * a

## Warning:  NAs produced by integer overflow

## [1] NA
```

Real numbers (or *floating points*) use a minimum of 4 bytes, for single precision floating points. In general 8 bytes are used to represent real numbers on a computer and these are called *double precision floating points* or *doubles*. Let's see some examples in R of how much space different types of variables take up.

Let's see how this plays out in terms of memory use in R.

```r
doubleVec <- rnorm(1e+05)
intVec <- 1:1e+05
set.seed(0)
charVec <- sample(letters, 1e+05, replace = TRUE)
object.size(doubleVec)

## 800040 bytes

object.size(intVec)  # so how many bytes per integer in R?

## 400040 bytes

object.size(charVec)

## 801288 bytes

.Internal(inspect(charVec))  # anything jump out at you?

## @3b17c10 16 STRSXP g0c7 [MARK,NAM(2)] (len=100000, tl=0)
##   @e5f9a8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "x"
##   @13db578 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "g"
##   @135b9f8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "j"
##   @1248948 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "o"
##   @e5f9a8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "x"
##   ...
```

We can easily calculate the number of megabytes (Mb) a vector of floating points (in double precision) will use as the number of elements times 8 (bytes/double) divided by 10^6 to convert from bytes to megabytes. (In some cases when considering computer memory, a megabyte is $1,048,576 = 2^{20} = 1024^2$ bytes so slightly different than 10^6). Finally, R has a special object that tells us about the characteristics of computer numbers on the machine that R is running on called *.Machine*. For example, *.Machine$integer.max* is $2147483647 = 2^{31} - 1$, which confirms how many bytes R is using for each integer (and that R is using a bit for the sign of the integer). We have $2 \cdot 2^{31} = 2^{32} = (2^8)^4$, i.e., 4 bytes, with each byte having 8 bits.

2 Floating point basics

2.1 Representing real numbers

Reals (also called floating points) are stored on the computer as an approximation, albeit a very precise approximation. As an example, with a double, the error in the distance from the earth to the sun is around a millimeter. However, we need to be very careful if we're trying to do a calculation that produces a very small (or very large number) and particularly when we want to see if numbers are equal to each other.

```
0.3 - 0.2 == 0.1
```

```
## [1] FALSE
```

```
0.3
```

```
## [1] 0.3
```

```
0.2
```

```
## [1] 0.2
```

```
0.1   # Hmmm...
```

```
## [1] 0.1
```

```
options(digits = 22)
a <- 0.3
b <- 0.2
a
```

```
## [1] 0.299999999999999888978

b

## [1] 0.200000000000000111022

a - b

## [1] 0.0999999999999997779554

0.1

## [1] 0.100000000000000055511

1/3

## [1] 0.333333333333333148296
```

Notice that we can represent the result accurately only up to the 16th decimal place. This suggests no need to show more than 16 decimal places and no need to print out any more when writing to a file. And of course, often we don't need anywhere near that many. *Machine epsilon* is the term used for indicating the accuracy of real numbers and it is defined as the smallest float, x, such that $1 + x \neq 1$:

```
1e-16 + 1

## [1] 1

1e-15 + 1

## [1] 1.00000000000001110223

2e-16 + 1

## [1] 1.00000000000000222045

.Machine$double.eps

## [1] 2.22044604925031308084e-16
```

Floating point refers to the decimal point (or radix point since we'll be working with base 2 and

decimal relates to 10). Consider Avogadro's number in terms of scientific notation: $+6.023 \times 10^{23}$. A real number on a computer is stored in what is basically scientific notation:

$$\pm d_0.d_1 d_2 \ldots d_p \times b^e \tag{1}$$

where b is the base, e is an integer and $d_i \in \{0, \ldots, b-1\}$. First, we need to choose the number of bits to represent e so that we can represent sufficiently large and small numbers. Second we need to choose the number of bits, p, to allocate to $d = d_1 d_2 \ldots d_p$, which determines the accuracy of any computer representation of a real. The great thing about floating points is that we can represent numbers that range from incredibly small to very large while maintaining good precision. The floating point floats to adjust to the size of the number. Suppose we had only three digits to use and were in base 10. In floating point notation we can express $0.12 \times 0.12 = 0.0144$ as $(1.20 \times 10^{-1}) \times (1.20 \times 10^{-1}) = 1.44 \times 10^{-2}$, but if we had fixed the decimal point, we'd have $0.120 \times 0.120 = 0.014$ and we'd have lost a digit of accuracy.

More specifically, the actual storage of a number on a computer these days is generally as a double in the form:

$$(-1)^S \times 1.d \times 2^{e-1023} = (-1)^S \times 1.d_1 d_2 \ldots d_{52} \times 2^{e-1023}$$

where the computer uses base 2, $b = 2$, because base-2 arithmetic is faster than base-10 arithmetic. The leading 1 normalizes the number; i.e., ensures there is a unique representation for a given computer number. This avoids representing any number in multiple ways, e.g., either $1 = 1.0 \times 2^0 = 0.1 \times 2^1 = 0.01 \times 2^2$. For a double, we have 8 bytes=64 bits. Consider our representation as (S, d, e) where S is the sign. The leading 1 is the *hidden bit*. In general e is represented using 11 bits ($2^{11} = 2048$), and the subtraction takes the place of having a sign bit for the exponent. This leaves $p = 52$ bits for d.

Question: Given a fixed number of bits for a number, what is the tradeoff between using bits for the d part vs. bits for the e part?

Let's consider what can be represented exactly:

```
0.1
```

```
## [1] 0.10000000000000000555
```

```
0.5
```

```
## [1] 0.5
```

```
0.25
```

```
## [1] 0.25
```

```
0.26
```

```
## [1] 0.26000000000000088818
```

```
1/32
```

```
## [1] 0.03125
```

```
1/33
```

```
## [1] 0.030303030303030387138
```

So why is 0.5 stored exactly and 0.1 not stored exactly? By analogy, consider the difficulty with representing 1/3 in base 10.

2.2 Overflow and underflow

The largest and smallest numbers we can represent are $2^{e_{\max}}$ and $2^{e_{\min}}$ where $e_{\max}$ and $e_{\min}$ are the smallest and largest possible values of the exponent. Let's consider the exponent and what we can infer about the range of possible numbers. With 11 bits for e, we can represent $\pm 2^{10} = \pm 1024$ different exponent values (see *.Machine$double.max.exp*) (why is *.Machine$double.min.exp* only -1022?). So the largest number we could represent is 2^{1024}. What is this in base 10?

```
log10(2^1024)  # whoops ... we've actually just barely overflowed
```

```
## [1] Inf
```

```
log10(2^1023)
```

```
## [1] 307.9536855642527370946
```

We could have been smarter about that calculation: $\log_{10} 2^{1024} = \log_2 2^{1024} / \log_2 10 = 1024/3.32 \approx 308$. Analogously for the smallest number, so we have that floating points can range between 1×10^{-308} and 1×10^{308}. Take a look at *.Machine$double.xmax* and *.Machine.double.xmin*. Producing something larger or smaller in magnitude than these values is called overflow and underflow respectively. When we overflow, R gives back an Inf or -Inf (and in other cases we might get an error message). When we underflow, we get back 0, which in particular can be a problem if we try to divide by the value.

2.3 Integers or floats?

Values stored as integers should overflow if they exceed *.Machine$integer.max*.

Should 2^{45} overflow?

```
x <- 2^45
z <- 25
class(x)

## [1] "numeric"

class(z)

## [1] "numeric"

as.integer(x)

## Warning:  NAs introduced by coercion

## [1] NA

as.integer(z)

## [1] 25

1e308

## [1] 1.0000000000000000010979e+308

as.integer(1e308)

## Warning:  NAs introduced by coercion

## [1] NA

1e309

## [1] Inf
```

In R, numbers are generally stored as doubles. We've basically already seen why - consider the maximum integer when using 4 bytes and the maximum floating point value. Representing integers as floats isn't generally a problem, in part because integers will be stored exactly in base

two provided the absolute value is less than 2^{53}. Why 2^{53}?

However, you can force storage as integers in a few ways: values generated based on *seq()*, based on the : operator, specified with an "L", or explicitly coerced:

```
x <- 3
typeof(x)

## [1] "double"

x <- as.integer(3)
typeof(x)

## [1] "integer"

x <- 3L
typeof(x)

## [1] "integer"
```

2.4 Precision

Consider our representation as (*S, d, e*) where we have $p = 52$ bits for d. Since we have $2^{52} \approx 0.5 \times 10^{16}$, we can represent about that many discrete values, which means we can accurately represent about 16 digits (in base 10). The result is that floats on a computer are actually discrete (we have a finite number of bits), and if we get a number that is in one of the gaps (there are uncountably many reals), it's approximated by the nearest discrete value. The accuracy of our representation is to within 1/2 of the gap between the two discrete values bracketing the true number. Let's consider the implications for accuracy in working with large and small numbers. By changing e we can change the magnitude of a number. So regardless of whether we have a very large or small number, we have about 16 digits of accuracy, since the absolute spacing depends on what value is represented by the least significant digit (the *ulp*, or *unit in the last place*) in d, i.e., the $p = 52$nd one, or in terms of base 10, the 16th digit. Let's explore this:

```
options(digits = 22)
.1234123412341234

## [1] 0.12341234123412339607 21
```

```
1234.1234123412341234 # not accurate to 16 places

## [1] 1234.123412341234143241

123412341234.123412341234 # only accurate to 4 places

## [1] 123412341234.1234130859

1234123412341234.123412341234 # no places!

## [1] 1234123412341234

12341234123412341234 # fewer than no places!

## [1] 12341234123412340736
```

We can see the implications of this in the context of calculations:

```
1234567812345678 - 1234567812345677

## [1] 1

12345678123456788888 - 12345678123456788887

## [1] 0

12345678123456780000 - 12345678123456770000

## [1] 10240

.1234567812345678 - .1234567812345677

## [1] 9.714451465470119728707e-17

.12345678123456788888 - .12345678123456788887

## [1] 0

.000012345678123456788888 - .000012345678123456788887

## [1] 0
```

Suppose we try this calculation: $123456781234 - .0000123456781234$. How many decimal places do we expect to be accurate?

The spacing of possible computer numbers that have a magnitude of about 1 leads us to another definition of *machine epsilon* (an alternative, but essentially equivalent definition to that given previously in this Unit). Machine epsilon tells us also about the relative spacing of numbers. First let's consider numbers of magnitude one. The difference between $1 = 1.00...00 \times 2^0$ and $1.000...01 \times 2^0$ is $1 \times 2^{-52} \approx 2.2 \times 10^{-16}$. Machine epsilon gives the *absolute spacing* for numbers near 1 and the *relative spacing* for numbers with a different order of magnitude and therefore a different absolute magnitude of the error in representing a real. The relative spacing at x is

$$\frac{(1+\epsilon)x - x}{x} = \epsilon$$

since the next largest number from x is given by $(1+\epsilon)x$. Suppose $x = 1 \times 10^6$. Then the absolute error in representing a number of this magnitude is $x\epsilon \approx 2 \times 10^{-10}$. (Actually the error would be one-half of the spacing, but that's a minor distinction.) We can see by looking at the numbers in decimal form, where we are accurate to the order 10^{-10} but not 10^{-11}.

```
1000000.1
```

```
## [1] 1000000.099999999976717
```

Let's see what arithmetic we can do exactly with integers stored as doubles and how that relates to the absolute spacing of numbers we've just seen:

```
2^52
```

```
## [1] 4503599627370496
```

```
2^52 + 1
```

```
## [1] 4503599627370497
```

```
2^53
```

```
## [1] 9007199254740992
```

```
2^53 + 1
```

```
## [1] 9007199254740992
```

```
2^53 + 2
```

```
## [1] 9007199254740994
```

```
2^54
```

```
## [1] 18014398509481984
```

```
2^54 + 2
```

```
## [1] 18014398509481984
```

```
2^54 + 4
```

```
## [1] 18014398509481988
```

The absolute spacing is $x\epsilon$, so $2^{52} \times 2^{-52} = 1$, $2^{53} \times 2^{-52} = 2$, $2^{54} \times 2^{-52} = 4$.

With a bit more work (e.g., using Mathematica), one can demonstrate that doubles in R in general are represented as the nearest number that can stored with the 64-bit structure we have discussed and that the spacing is as we have discussed. The results here show the spacing that results, in base 10, for numbers around 1. The numbers R reports are spaced in increments of individual bits in the base 2 representation.

```
options(digits = 22)
0.1234567812345678
```

```
## [1] 0.1234567812345677972896
```

```
0.12345678123456781
```

```
## [1] 0.1234567812345678111674
```

```
0.12345678123456782
```

```
## [1] 0.1234567812345678250452
```

```
0.12345678123456783
```

```
## [1] 0.1234567812345678250452
```

```
0.12345678123456784
```

```
## [1] 0.1234567812345678389 23
```

2.5 Working with higher precision numbers

The *Rmpfr* package allows us to work with numbers in higher precision. (This code is not working with *knitr*, so I'm just showing the code here, not the output.)

```
require(Rmpfr)
piLong <- Const("pi", prec = 260)  # pi 'computed' to correct 260-bit preci
piLong  # nicely prints 80 digits
mpfr(".1234567812345678", 40)
mpfr(".1234567812345678", 80)
mpfr(".1234567812345678", 600)
```

In contrast to R, Python has arbitrary precision integers. So, e.g., `pow(3423333, 15)` returns an integer. But floating points are handled in similar fashion to R.

3 Implications for calculations and comparisons

3.1 Computer arithmetic is not mathematical arithmetic!

As mentioned for integers, computer number arithmetic is not closed, unlike real arithmetic. For example, if we multiply two computer floating points, we can overflow and not get back another computer floating point. One term that is used, which might pop up in an error message (though probably not in R) is that an "exception" is "thrown". Another mathematical concept we should consider here is that computer arithmetic does not obey the associative and distribute laws, i.e., $(a+b)+c$ may not equal $a+(b+c)$ on a computer and $a(b+c)$ may not be the same as $ab+ac$. Here's an example:

```
val1 <- 1/10
val2 <- 0.31
val3 <- 0.57
res1 <- val1 * val2 * val3
res2 <- val3 * val2 * val1
identical(res1, res2)

## [1] FALSE

res1

## [1] 0.017669999999999982081
```

```
res2
```

```
## [1] 0.01767000000000000167755
```

3.2 Calculating with integers vs. floating points

It's important to note that operations with integers are fast and exact (but can easily overflow) while operations with floating points are slower and approximate. Because of this slowness, floating point operations (*flops*) dominate calculation intensity and are used as the metric for the amount of work being done - a multiplication (or division) combined with an addition (or subtraction) is one flop. We'll talk a lot about flops in the next unit on linear algebra.

3.3 Comparisons

As we saw, we should never test `a==b` unless (1) *a* and *b* are represented as integers in R, (2) they are integers stored as doubles that are small enough that they can be stored exactly) or (3) they are decimal numbers that have been created in the same way (e.g., `0.1==0.1` vs. `0.1==0.4-0.3`. Similarly we should be careful about testing `a==0`. And be careful of greater than/less than comparisons. For example, be careful of `x[ x < 0 ] <- NA` if what you are looking for is values that might be *mathematically* less than zero, rather than whatever is *numerically* less than zero.

```
4L - 3L == 1L
```

```
## [1] TRUE
```

```
4 - 3 == 1
```

```
## [1] TRUE
```

```
4.1 - 3.1 == 1
```

```
## [1] FALSE
```

One nice approach to checking for approximate equality is to make use of *machine epsilon*. If the relative spacing of two numbers is less than *machine epsilon*, then for our computer approximation, we say they are the same. Here's an implementation that relies on the absolute spacing being $x\epsilon$ (see above):

`if(abs(a - b) < .Machine$double.eps * abs(a + b)) print("approximately equal")`

Actually, we probably want to use a number slightly larger than *.Machine$double.eps* to be safe. You can also take a look at the R function *all.equal.numeric()*.

Finally, in computing, we often encounter the use of an unusual integer as a symbol for missing values. E.g., a datafile might store missing values as -9999. Testing for this using == in R should generally be ok: `x [ x == -9999 ] <- NA`, but only because integers of this magnitude are stored exactly. To be really careful, you can read in as character type and do the assessment before converting to numeric.

3.4 Calculations

Given the limited *precision* of computer numbers, we need to be careful when:

- Subtracting large numbers that are nearly equal (or adding negative and positive numbers of the same magnitude). You won't have the precision in the answer that you would like.

```
123456781234.56 - 123456781234.00

## [1] 0.55999755859375
```

The absolute error here, based on the larger value (which has the fewest error-free decimal places) is of the order $\epsilon x = 2.2 \times 10^{-16} \cdot 1 \times 10^{12} \approx 1 \times 10^{-4} = .0001$, so while we might think that the result is close to the value 1 and should have error of about machine epsilon, we actually only have about four significant digits in our result.

This is called *catastrophic cancellation*, because most of the digits that are left represent rounding error - all the significant digits have cancelled with each other.
Here's catastrophic cancellation with small numbers. The right answer here is exactly 0.000000000000000000001234.

```
a = .000000000000123412341234

b = .000000000000123412340000

a - b

## [1] 1.2339999931513974908 51e-21
```

But the result is accurate only to 8 places + 21 = 29 places, as expected from a machine precision-based calculation, since the "1" is in the 13th position (13+16=29). Ideally, we would have accuracy to 37 places (16 + the 21), but we've lost 8 digits to catastrophic cancellation.

It's best to do any subtraction on numbers that are not too large. For example, we can get catastrophic cancellation when computing a sum of squares in a naive way:

$$s^2 = \sum x_i^2 - n\bar{x}^2$$

```
x <- c(-1, 0, 1) + 1e8

n <- length(x)

sum(x^2)-n*mean(x)^2 # that's not good!

## [1] 0

sum((x - mean(x))^2)

## [1] 2
```

A good principle to take away is to subtract off a number similar in magnitude to the values (in this case $\bar{x}$ is obviously ideal) and adjust your calculation accordingly. In general, you can sometimes rearrange your calculation to avoid catastrophic cancellation. Another example involves the quadratic formula for finding a root (p. 101 of Gentle).

- Adding or subtracting numbers that are very different in magnitude. The precision will be that of the large magnitude number, since we can only represent that number to a certain absolute accuracy, which is much less than the absolute accuracy of the smaller number:

```
123456781234 - 1e-06

## [1] 123456781234
```

The absolute error in representing the larger number is around 1×10^{-4} and the smaller number is smaller than this.

A work-around is to add a set of numbers in increasing order. However, if the numbers are all of similar magnitude, then by the time you add ones later in the summation, the partial sum will be much larger than the new term. A work-around is to add the numbers in a tree-like fashion, so that each addition involves a summation of numbers of similar size.

Given the limited *range* of computer numbers, be careful when you are:

- Multiplying or dividing many numbers, particularly large or small ones. Never take the product of many large or small numbers as this can cause over- or under-flow. Rather compute on the log scale and only at the end of your computations should you exponentiate. E.g.,

$$\prod_i x_i / \prod_j y_j = \exp(\sum_i \log x_i - \sum_j \log y_j)$$

- Challenge: Let's think about how we can handle the following calculation. Suppose I want to calculate a predictive density (e.g., in a model comparison in a Bayesian context):

$$\begin{aligned} f(y^*|y,x) &= \int f(y^*|y,x,\theta)\pi(\theta|y,x)d\theta \\ &\approx \frac{1}{M}\sum_{j=1}^{m}\prod_{i=1}^{n} f(y_i^*|x,\theta_j) \\ &= \frac{1}{M}\sum_{j=1}^{m}\exp\sum_{i=1}^{n}\log f(y_i^*|x,\theta_j) \\ &\equiv \frac{1}{M}\sum_{j=1}^{m}\exp(v_j) \end{aligned}$$

First, why do I use the log conditional predictive density? Second, let's work with an estimate of the unconditional predictive density on the log scale, $\log f(y^*|y,x) \approx \log \frac{1}{M}\sum_{j=1}^{m}\exp(v_j)$. Now note that e^{v_j} may be quite small as v_j is the sum of log likelihoods. So what happens if we have terms something like e^{-1000}? So we can't exponentiate each individual v_j. Thoughts? I have one solution in mind, but there might be other approaches.

Numerical issues come up frequently in linear algebra. For example, they come up in working with positive definite and semi-positive-definite matrices, such as covariance matrices. You can easily get negative numerical eigenvalues even if all the eigenvalues are positive or non-negative. Here's an example where we use an squared exponential correlation as a function of time (or distance in 1-d), which is *mathematically* positive definite [all the eigenvalues are positive] but not numerically positive definite:

```
require(fields)
xs <- 1:100
dists <- rdist(xs)
corMat <- exp(-(dists/10)^2)
eigen(corMat)$values[80:100]
```

```
##  [1]  2.025087032040071293067e-18 -3.266419741215397140920e-17
##  [3] -3.444200677004415898082e-17 -4.886954483578307325434e-17
##  [5] -6.129347918638579386910e-17 -9.880603825772825889419e-17
##  [7] -9.967343900132262641741e-17 -1.230143695483269682612e-16
##  [9] -1.248024408367381367725e-16 -1.292974005668460125397e-16
## [11] -1.331124664942472191787e-16 -1.651346230025272135916e-16
## [13] -1.951061360969111230889e-16 -1.990648753104187720567e-16
## [15] -2.015924870201480734054e-16 -2.257013487287240792239e-16
## [17] -2.335683529300324037256e-16 -2.719929490669187250991e-16
## [19] -2.882703020809833099805e-16 -3.057847173103147957185e-16
## [21] -4.411825302647757790411e-16
```

3.5 Final note

How the computer actually does arithmetic with the floating point representation in base 2 gets pretty complicated, and we won't go into the details. These rules of thumb should be enough for our practical purposes. Monahan and the URL reference have many of the gory details.