MT 4113, Computing in Statistics

Lecture 4 - An introduction to computer arithmetic

Len Thomas

Oct 1 2018

1 An Introduction to Computer Arithmetic

1.1 Introduction

About this lecture

This is a rough guide to how computers store and manipulate numbers[1]

Motivating example using `R`

What does this code do?

```
x <- 0.1
x <- x + 0.05
x

if (x == 0.15) {
  cat("Len is a great guy!\n")
} else {
  cat("Len is a loser!\n")
}
```

Numbers on computers

- Computers really can't do arithmetic. They recognize two states: on (1) and off (0). Numbers on computers are really a collection of binary digits ('bits') – they only have meaning by human convention.
- Numbers are represented on computers in two ways:
  - fixed point (integers)
  - floating point (reals)

Definitions: Data size

- bit: short for binary digit smallest unit of storage on a machine – holds either 0 or 1
- byte: short for bynary term [2] 8 bits – enough to hold a single character of text (see later)
- word: natural data size of a computer. Depends on the CPU – for most desktop systems currently it is 64 bits (or 32 bits for older machines).
- kilobyte (K): $2^{10} = 1,024$ bytes
- megabyte (MB): $2^{20} = 1,048,576$ bytes$= 1,024$ K
- gigabyte (GB): $2^{30} = 1,073,741,824$ bytes[3] $= 1,024$ MB

[1] A much more in-depth treatment is in, e.g., Chapter 2 of Gentle, J.E. Computational Statistics, available from the library as an electronic book.

[2] Computer scientists can't spell!

[3] Hard disk drive manufacturers sometimes prefer to use $1,000,000,000$ bytes as it makes their drives look bigger.

1.2 Fixed point

[fragile]

Fixed point representations

- Example representation: signed integer[4]
  - 1st bit is the sign (0=+ve, 1=-ve)
  - Remaining bits are the absolute value in base 2
- E.g., on a 4-bit system:

0000 is 0	1000 is -0
0001 is 1	1001 is -1
0010 is 2	1010 is -2
0011 is 3	1011 is -3
⋮	⋮
0111 is 7	1111 is -7

Fixed point arithmetic - addition

- E.g. addition:

3	0011	
+2	0010	
5	0101	add

Fixed point arithmetic - multiplication

- Multiply by a power of 2 - just shift the bits left and fill in rightmost bits with zeros

3	0011	
x2	0010	
6	0110	x2 ⇒ shift left one

- If not a power of 2, use the 'shift and add' algorithm: express as a sum of powers of 2. E.g., $2 \times 3 = (2 \times 2) + (2 \times 1)$

2	0010	
x3	0011	
4	0100	x2 ⇒ shift left one
2	0010	x1 ⇒ shift left none
6	0110	add

Fixed point arithmetic - subtraction and division

- Dividing by a power of two is easy
  - just shift right (and drop any fractional part)
- Otherwise, dividing and subtracting are not very intuitive (or fast) in the signed integer representation, so we do not cover them explicitly here
- They are better (easier) in other representations – e.g., 1s or 2s complement

[4] there are other more efficient representations, e.g., 1s complement and 2s complement

Fixed point overflow

$$\begin{array}{rr} 3 & 0011 \\ +5 & 0101 \\ \hline 0 & 0000 \end{array}$$

- Machine needs a protocol to deal with and warn user of fixed point overflow

Fixed point numbers in R

- In R, the fixed point data type is called `integer`.
- Integers aren't actually used much in R, except in loops and indexing elements in arrays.
- Integers in R are stored in 32 bits.

```
> as.integer(2 ^ 31 - 1)
[1] 2147483647
> as.integer(2 ^ 31 - 1) + as.integer(1)
[1] NA
Warning message:
In as.integer(2 ^ 31 - 1) + as.integer(1) : NAs produced by integer overflow
```

Summary: Fixed point representations

- Advantages:
  - Results are exact
  - Fast (compared with floating point)
- Disadvantages:
  - Limited applicability: integers and related (e.g., financial applications, dates[5])
  - Limited range

1.3 Floating point

Floating point representations

- Example $S.F \times B^E$ e.g., $+.1 \times 10^{+1}$
  - S = sign
  - F = fraction – unsigned integer. Typically 'normalized' so that 1st digit is non-zero.
  - B = base
  - E = exponent – signed integer.
- On a computer
  - B is 2
  - S is 1 bit (0=+, 1=-)
  - F and E are allocated a fixed number of bits each
- For example, consider an 8-bit floating point representation with
  - 1 bit for the sign
  - 3 for the exponent (a signed integer)
  - 4 for the fraction (an unsigned integer)

		decimal		binary		sign	exponent	fraction
1	=	$+.1 \times 10^{+1}$	=	$+.1_{\text{two}} \times 2^{+1}$	=	0	001	1000
3	=	$+.3 \times 10^{+1}$	=	$+.11_{\text{two}} \times 2^{+2}$	=	0	010	1100
$-1/8$	=	$-.125 \times 10^{+0}$	=	$-.1_{\text{two}} \times 2^{-2}$	=	1	110	1000

[5] Dates can use unsigned integers representations

Limitations

- Given that the exponent and fraction are stored in a fixed number of bits, there are limitations both to the size and accuracy with which numbers can be stored
- Size: in our example
  - Largest possible number is
  - $+.1111_{\text{two}} \times 2^3 = 7.5$
  - Smallest possible positive number[6] is
  - $+.0001_{\text{two}} \times 2^{-3} = 1/128 = 0.0078125$
- Accuracy:

	binary	sign	exponent	fraction
1	$.1000_{\text{two}} \times 2^{+1}$	0	001	1000
$+1/128$	$+.0001_{\text{two}} \times 2^{-3}$	0	111	0001
1	$.1000_{\text{two}} \times 2^{+1}$	0	001	1000

  - to see why, re-express $1/128$ as

$+1/128$	$+.00000001_{\text{two}} \times 2^{+1}$	0	001	00000001

- Let ϵ be the smallest positive number, such that $\epsilon + 1.0 \neq 1.0$
- In our example
  - $\epsilon = 1/8$:

	binary	sign	exponent	fraction
1	$.1000_{\text{two}} \times 2^{+1}$	0	001	1000
$+1/8$	$+.1000_{\text{two}} \times 2^{-2}$	0	110	1000
1 1/8	$.1001_{\text{two}} \times 2^{+1}$	0	001	1001

  - to see why, re-express $1/8$ as

$+1/8$	$+.0001_{\text{two}} \times 2^{+1}$	0	001	0001

- In general, ϵ, the 'machine epsilon' (or closely related 'machine unit'), governs the relative accuracy of calculations (see later for an example). $\epsilon = 2^{(1-d)}$ where d is the number of bits in the fraction.

Some consequences of lack of accuracy

For clarity, the examples here use a base 10 representation with 4 decimal places in the fraction.

- Things rarely add up exactly, due to rounding errors.

$$\begin{aligned} 1/3 + 1/3 + 1/3 &= 1 \\ .3333 + .3333 + .3333 &= .9999 \end{aligned}$$

- Order of summation matters (associative rule doesn't work). E.g.:

$$1.000 + .0001 + \{9{,}998 \text{ more } .0001\text{s}\} + .0001 = 1.000$$
$$.0001 + \{9{,}998 \text{ more } .0001\text{s}\} + .0001 + 1.000 = 2.000$$

- When adding n values to x, the maximum error $< \frac{n\epsilon x}{2}$.
- Mixed signs can cause problems – 'catastrophic cancellation': subtraction of two nearly equal numbers eliminates the most significant digits, and any small rounding errors that were previously hidden end up becoming important.
  - Poor example: $.1000 \times 10^0 - .9999 \times 10^{-1} = .1000 \times 10^{-3}$ instead of $.1000 \times 10^{-4}$ – so the answer is wrong in what is now the most significant digit.
  - Better example: calculating the sample variance using the usual $s^2 = \frac{\sum(x^2)}{n} - \left(\frac{\sum x}{n}\right)^2$ – can get -ve estimates! If you want to see, try it for the sample $\{356, 357, 358, 359, 360\}$ keeping 4 decimal places in the fraction.

[6] if we allow the leading digits of the fraction to be zero ('de-normalized')

Floating point numbers in R

- In R, the floating point data type is called `numeric`.
- It is the default for numbers in R.

```
> x <- 3
> class(x)
[1] "numeric"
```

- `numeric` values are represented using an IEEE standard for double precision. 64 bits in total, 1 bit for the sign, 11 bits for the exponent, 53 for the the fraction (uses a trick to gain 1 extra bit).
- So,
  - the largest possible number is[7] $+.1111\ldots_{\text{two}} \times 2^{+1023} = 8.98 \times 10^{+307}$

    ```
    > 1 * 2 ^ 1023
    [1] 8.988466e+307
    > 2 * 2 ^ 1023
    [1] Inf
    ```

  - accuracy: machine epsilon is[8] $\epsilon = 2^{(1-53)} = 2.20 \times 10^{-16}$

    ```
    > 1 + 2 ^ (1 - 53) == 1
    [1] FALSE
    > 1 + (0.5 * 2 ^ (1 - 53)) == 1
    [1] TRUE
    ```

- The standard dictates how some arithmetic operations should be done, and defines special values +Inf and -Inf (for overflows), and NaN (not a number).

```
> 1 / 0
[1] Inf
> 0 / 0
[1] NaN
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> Inf / Inf
[1] NaN
```

- Underflows are set to 0, and no warning is generated.

```
> exp(-1000)
[1] 0
```

Living with floating point limitations

- Re-work expressions and algorithms to
  - Avoid generating very large numbers that might overflow. Underflow is generally better than overflow. E.g. Rewrite $\frac{e^x}{1+e^x}$ as $\frac{1}{1+e^{-x}}$

[7] it is slightly higher, due to some tricks
[8] again, it's slightly lower, due to some tricks

```
> x <- 1000
> exp(x) / (1 + exp(x))
[1] NaN
> 1 / (1 + exp(-x))
[1] 1
```

  - Minimize large differences in scale in intermediate calculations, so as to preserve accuracy. For example, instead of taking the product of a large number of likelihoods, take the sum of the log likelihoods and then exponentiate.

- Normalize[9] – scale values so they are centred near 0, and perhaps divide by some suitably large value so they don't overflow. Allows the maximum range of values.
- Change exact tests to 'good enough' tests. From `if(x == y) then...` to `if(((x - e) <= y) & (x + e >= y)) then ...` or, better (scale independent) `if(abs(x - y) <= abs(y) * e) then ...`
- Be careful about the order of operations - e.g., consider sorting numbers before adding (there are even smarter tricks here).
- If you're desperate, use double-precision (at least for intermediate calculations) – but it uses twice the memory and can use 3-4 times the time. (Although this does depend on the size of the floats relative to computer word size, so best to try it and see.)[10]

Some comments:

- Note that many of these show a trade-off between accuracy and speed.
- So, first ask yourself if it's going to be a problem, before spending lots of time fixing a problem that isn't important!

Summary: floating point representations

- Advantages:
  - Wide applicability (all numbers on the real line)
  - Wider range than fixed point
- Disadvantages:
  - Results are not exact – need to be careful
  - Slow (compared with fixed point)

Motivating example Redux

What should we do instead of this?

```
if (x == 0.15) {
  cat("Len is a great guy!\n")
} else {
  cat("Len is a loser!\n")
}
```

[9] Note: different meaning here from normalizing the fractional part of a floating point number

[10] Doesn't apply to R, which always uses double precision. If you want even more accuracy, you can use the `Rmpfr` package.

1.4 Character storage

Character storage[11]

- Characters are stored as binary numbers on the computer. The correspondence between the stored numbers and the displayed symbols is determined by a character set.
- Example character sets:
  - ASCII (American Standard Code for Information Exchange). A 7-bit system, so there are 128 unique characters. 0-31 are non-printing control characters (e.g. 27 is ESCape), the other 96 are alphanumeric characters and symbols (e.g. 63 is ?, 77 is M).
  - Extended ASCII. A set of 8-bit systems - the extra 128 characters are used for European accents, etc. There are many regional variants. Used in old versions of windows, and (I think) in R.
  - UNICODE. A 16-bit system, so there are 65,536 characters. Designed to include extended-character languages like Chinese and Japanese - but not really sufficient for that. Used in MS Windows (2000 and later).
- Practical note: there can be hassles switching between operating systems or locales within operating systems. E.g., Unix (LF) to Windows (CR+LF).

Practice questions

1. Write out -15-3=-18 in binary using an 8-bit signed integer system
2. What is the largest positive number that can be represented in this system?
3. Why is it very fast to multiply numbers represented as integers by 2? What manipulation is required to the bits making up the number to achieve this operation?
4. What is the machine epsilon on a 32-bit floating point system where 1 bit is for the sign, 16 bits for the exponent and 16 bits[12] for the fraction?
5. In this system, what would be the result of computing $2 * 2^{-17} + 1$?
6. What do you get if you compute $512 * 0.25$ in the above floating point system, and then convert it into the above signed integer system?[13]

[11] OK - I know characters aren't numbers but I thought I'd include this stuff anyway

[12] using the same trick as in the IEEE standard to get one extra bit for free; note that this is not a very sensible allocation as it gives far too much of the space to the exponent

[13] A calculation like this once cost the EU space program about $500 million: https://www.ima.umn.edu/~arnold/disasters/ariane.html