Numerical Analysis
Mathematical Description of Computer Calculations

Some Numerical Considerations

Time for a little numerical theory:

- Computer memory is a long series of switches that can be 'on' or 'off'.
- One switch = 1 bit, 8 bits = 1 byte.
- Numbers are stored in binary arithmetic:

$$0 = 0,\ 1 = 1,\ 2 = 10,\ 3 = 11,\ 11 = 1011$$

1011 in binary means

$$1 * 8 + 0 * 4 + 1 * 2 + 1 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$

just as 11 in decimal arithmetic means

$$1 * 10^1 + 1 * 10^0$$

Computer Rules for Arithmetic

Adding 123 (base 10) = 1111011 (base 2) to 14 (base 10) = 1110 (base 2).

123	0	1	1	1	1	0	1	1
14	0	0	0	0	1	1	1	0
Memory	1	1	1	1	1	1	0	0
Result	1	0	0	0	1	0	0	1

Memory = "carry the 1" as in adding in decimal.

From right column, change Result and (Memory in next column) left with rules

# 1's in 123, 14, Memory	(Result, Memory next column left)
0	(0,0)
1	(1,0)
2	(0,1)
3	(1,1)

Storing Numbers in Binary

- Normally, numbers are stored in 8 bytes = 64 bits.
- But, first bit must be used for the sign (on (1) = positive).
- So 11 looks like

1	0	0	0	6 * 0 bytes	1	0	1	1

- Largest integer = 2,147,483,648; restricts what numbers are computable.
- Two ways of storing zero! Alternatively, store numbers *relative to* 2^{63} ($11 = 2^{63} - (2^{63} - 11)$ so store the $(2^{63} - 11)$).
- Other schemes also possible.
- But how do I store $1/2 = 2^{-1}$?

Floating Point Numbers

- Represent numbers as an integer (or a decimal) and a power

$$0.0023456 = 2.3456 * 10^{-3}, 543000 = 5.43 * 10^{5}$$

  Of course, this is done in binary (including the decimal point!)
- `R` expresses large (or small) numbers this way: `5.43e5` = 543,000 (always in decimals).
- Give over 2 bytes to the power:

sign (1 bit)	significand (53 bits)	exponent (8 bits)

- sign = positive or negative
- significand (also mantissa) = number: 2.3456 above
- exponent = power (-3 above)
- In binary, 11 = 1011 is then written as

1	1	0	1	1	0	...	0	0	0	0	0	0	0	1	1

Some Consequences

- What happens if my significand won't fit in 53 bits? Drop digits off the back.
- 64-bit floating point arithmetic $\approx$ calculating with 16 significant figures.
- Back in base 10: 0.0023456 with significand recorded to 4 digits $= 2.346 * 10^{-3}$
- "Round-off" error is -0.0000004 (in this toy case).
- Exponent stored as integer – largest value and smallest values are $\pm 2^{1023}$; translates to largest possible number $\approx 10^{307}$.
- So what?
  1. Testing statements
  2. Error Accumulation
  3. Overflow
- In `R`, you can find specification of numerical tolerances by typing `.Machine` at the prompt.

Floating Point Numbers and Logic

Tests of logical statements may not hold:

- $\sqrt{2} * \sqrt{2} = 2$, but $\sqrt{2}$ is approximated in floating point numbers; `R` will return `FALSE`
- $10^{16} + 10^{-16} \neq 10^{16}$, but `R` returns `TRUE` because it doesn't store enough digits on the right hand side.
- Need to specify these tests to within a tolerance; usually an order of magnitude larger than `.Machine$double.eps`.
- `R` function `all.equal` will test up to a given tolerance (defaults to `.Machine$double.eps`

```
> all.equal(sqrt(2)*sqrt(2),2)
[1] TRUE
```

Error Accumulation

- `.Machine$double.eps`: smallest ϵ such that $1 + \epsilon \neq 1$; normally around 10^{-16}.
- Important concepts: *relative* versus *absolute* error.
- If I want to use x, but can only use $x + \delta$,
  - Absolute error is δ.
  - Relative error is δ/x: amount relative to size of x.

  Round-off error is always relative to x.
- But also remember that R introduces errors in some of its calculations (eg division not represented directly).
- If you perform many operations, errors accumulate, especially if your numbers are small.
- Examples: operations on very large data sets, or on large matrices (see examples next).
- Some solutions: if your numbers are small (a data set I used recently had 5,000 records, all around 10^{-6}), try multiplying them first, perform the operation and divide appropriately.

Truncation Error Example

Consider the sequence

$$x_1 = 1,\ x_2 = \frac{1}{3},\ x_i = \frac{13}{3}x_{i-1} - \frac{4}{3}x_{i-2}$$

in fact, this generates $x_k = (1/3)^{k-1}$ for all k.

Certainly true for $k = 1$ and $k = 2$, suppose it's true for $k = j - 1$ and $k = j - 2$, then

$$\begin{aligned} x_j &= \frac{13}{3}\left(\frac{1}{3}\right)^{j-2} - \frac{4}{3}\left(\frac{1}{3}\right)^{j-3} \\ &= \left(\frac{1}{3}\right)^{j-1}\left[\frac{13}{3}\left(\frac{1}{3}\right)^{-1} - \frac{4}{3}\left(\frac{1}{3}\right)^{-2}\right] \\ &= \left(\frac{1}{3}\right)^{j-1}[13 - 12] \end{aligned}$$

Truncation Error Example

What happens if we code this?

```
> x = rep(0,18); x[1] = 1; x[2] = 1/3
> for(i in 3:18){ x[i] = (13/3)*x[i-1] - (4/3)*x[i-2] }
```

```
> x
 [1]  1.000000e+00  3.333333e-01  1.111111e-01  3.703704e-02  1.234568e-02
 [6]  4.115226e-03  1.371742e-03  4.572474e-04  1.524158e-04  5.080526e-05
[11]  1.693507e-05  5.644977e-06  1.881469e-06  6.263947e-07  2.057519e-07
[16]  5.639888e-08 -2.994080e-08 -2.049420e-07
```

And compare this with calculating it directly.

```
> (1/3)^(0:17)
 [1] 1.000000e+00 3.333333e-01 1.111111e-01 3.703704e-02 1.234568e-02
 [6] 4.115226e-03 1.371742e-03 4.572474e-04 1.524158e-04 5.080526e-05
[11] 1.693509e-05 5.645029e-06 1.881676e-06 6.272255e-07 2.090752e-07
[16] 6.969172e-08 2.323057e-08 7.743524e-09
```

Error `x[1]` is $1e^{-16}$, but that gets multiplied by 4.333 to calculate `x[2]`, and multiplied again for `x[3]`.

Meanwhile, the quantity that we're estimating gets smaller!

Further Round-Off Errors

- Represent relative error for a as $a(1+\epsilon)$, then relative error for ab is

$$a(1+\epsilon)b(1+\delta) = ab(1+\epsilon)(1+\delta) \approx ab(1+\epsilon+\delta)$$

  so relative errors add in multiplication.
- Addition adds *absolute* errors; usually doesn't change relative error.
- Book example: add 1,000 of size 10^6 each with absolute error 10^{-10}. Absolute error of sum is 10^{-7}, but sum is 10^9 so relative error is still the same.
- But consider two ways of taking an average:
  1. Sum everything and then divide by n: drop any digits we can't store in the sum.
  2. Iterate: $Y_{k+1} = \frac{k}{k+1} Y_k + \frac{1}{k+1} X_k$ output Y_n; numbers never get larger than the original X_i.

Catastrophic Cancelation

Subtraction can cause larger problems.

- Absolute error of $a - b$ is sum of absolute errors.
- Relative error, depends on how big $a - b$ is.
- If a and b both have relative error 10^{-16}, say, but $|a - b|/a$ is 10^{-6}, then the relative error of the difference is 10^{-10}.
- Book example:
  1,234,567,812,345,678 - 1,234,567,800,000,000 = 12,345,678.
  But floating point will round last digit of first number; error is 2. Size is 10^{-16} relative to first number, but 10^{-8} relative to output.
- Problems not always easily avoidable.

Example

Consider plotting the function

$$f(x) = \frac{1 - \cos(x)}{x^2}$$

close to $x = 0$.

- $\cos(x) \approx 1$ for $x \approx 0$, relative error of $1 - \cos(x)$ approaches 1.
- x^2 on denominator magnifies this effect.

```
x = seq(-4e-8,4e-8,8e-10)
z = (1-cos(x))/x^2
plot(x,z,type='l')
```

A Better Calculation

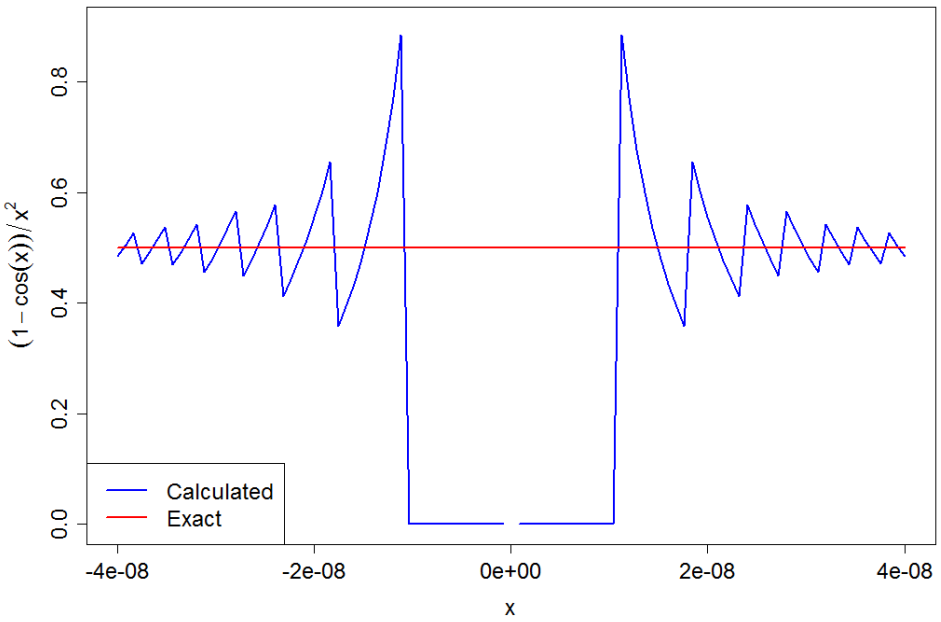
Taylor-series expansion of cosine:

$$\cos(x) = 1 - 0 \cdot x + \frac{1}{2}x^2 - 0 \cdot x^3 + \frac{1}{24}x^4 - 0 \cdot x^5 + o(x^6)$$

then

$$\frac{1 - \cos(x)}{x^2} = \frac{1}{2} + \frac{1}{24}x^2 + o(x^4)$$

```
z2 = 1/2 + x^4/24
lines(x,z2,type='l')
legend('bottomleft',
    c('Calculated','Exact'))
```



When $x = x(10^{-8})$,
$x^4 = o(10^{-32})$ and relative error is 10^{-24}.

Overflow and Underflow

1. Largest value storable is in `.Machine$double.xmax` ($1.797693 * 10^{308}$ on my laptop).
2. What if I add 10^{300} to it? `R` returns `Inf`.
3. But note that `.Machine$double.xmat + 1` is still $1.797693 * 10^{308}$.
4. This is defined as *numerical overflow*: we ask `R` to store a number that it cannot hold.
5. Same is true if you try to calculate a number that is too small (not quite `.Machine$double.xmin`), `R` returns `0`.)
6. In some numerical arithmetic, overflow can result in a form of wrapping and produce a very small number instead.

For most computations in this class, only testing numerical equalities will be relevant; but you should be aware of the potential for numerical problems to occur.

Inf, NaN, NA

R also contains three entries to indicate numerical problems:

Inf *Infinity*. Could be from numerical overflow, or exactly true. R will do sensible calculations with it; eg

```
> x = 10/0        > 5 - x          > pi/x
[1] Inf           [1] -Inf         [1] 0
```

NaN *Not a Number*. R declares this non-computable (try 0/0). Any calculation with NaN returns NaN.

```
> 5*NaN
[1] NaN
```

NA *Not Available*. Like NaN, but also exists for other data types; mostly used for missing entries in data (you can direct R to ignore these).

```
> 5*NA
[1] NA
```

Summary

- Numerical storage of numbers.
- Floating point arithmetic leads to round-off errors; inexact calculations can produce errors.
- Round-off errors become important in many repeated calculations, and especially in subtraction.
- Special entries for infinity and non-number calculations.