4a - Numerical Accuracy and Program Efficiency

Noud van Giersbergen

University of Amsterdam

2021-04-20

Motivation Numerical Analysis

- In mathematics, we have to deal with *continuous* problems, but computers can in principle only handle *discrete* items:
- Computers do not know real numbers, particularly no $\sqrt{2}$, no π, and no $1/3$, but only approximations of discretely (separately, thus not densely) numbers.
- Computers do not know functions such as the sine, but only know approximations consisting of simple components (e.g. polynomials).
- The magic word for the successful transition "continuous $\longrightarrow$ discrete" is called ***discretization***.
- We discretize real numbers by introduction of *floating point numbers*.

Numerical accuracy and program efficiency (spuRs Ch.9)

Machine representation of numbers

Computers use switches to encode information. A single ON/OFF indicator is called a bit; a group of eight bits is called a byte. Although it is quite arbitrary, it is usual to associate 1 with ON and 0 with OFF.

Integers

A fixed number of bytes is used to represent a single integer, usually four or eight. Let k be the number of bits we have to work with (usually 32 or 64). There are a number of schema used to encode integers.

Example: $k = 3$

Sign-and-Magnitude

111	-3
110	-2
101	-1
100	-0
000	0
001	1
010	2
011	3

Biased

000	-3
001	-2
010	-1
011	0
100	1
101	2
110	3
111	4

Two's complement

100	-4
101	-3
110	-2
111	-1
000	0
001	1
010	2
011	3

Sign-and-magnitude scheme

- Use one bit to represent the sign $+/-$ and the remainder to give a binary representation of the magnitude.
- Using the sequence $\pm b_{k-2} \cdots b_2 b_1 b_0$, where each b_i is 0 or 1, we represent the decimal number $\pm(2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \cdots + 2^{k-2} b_{k-2})$.
- For example, taking $k = 8$, -0100101 is interpreted as $-(2^5 + 2^2 + 2^0) = -37$. The smallest and largest integers we can represent under this scheme are $-(2^{k-1} - 1)$ and $2^{k-1} - 1$.
- The disadvantage of this scheme is that there are two representations of 0.

Biased scheme

- Use the sequence $b_{k-1} \cdots b_1 b_0$ to represent the decimal number $2^0 b_0 + 2^1 b_1 + \cdots 2^{k-1} b_{k-1} - (2^{k-1} - 1)$.

- For example, taking $k = 8$, 00100101 is interpreted as $37 - 127 = -90$.

- The smallest and largest integers represented under this scheme are $-(2^{k-1} - 1)$ and 2^{k-1} .

- The disadvantage of this scheme is that addition becomes a little more complex.

Two's complement scheme

- Given k bits, the numbers 0, $1, \ldots, 2^{k-1} - 1$ are represented in the usual way, using a binary expansion, but the numbers -1, $-2, \ldots, -2^{k-1}$ are represented by $2^k - 1$, $2^k - 2, \ldots, 2^k - 2^{k-1}$.
- Addition under this scheme is equivalent to addition modulo 2^k, and can be implemented very efficiently.
- The representation of integers on your computer happens at a fundamental level, and R has no control over it. The largest integer you can represent on your computer (whatever encoding scheme is in use) is known as *maxint*; R records the value of maxint on your computer in the variable `.Machine`.

```
.Machine$integer.max
```

```
[1] 2147483647
```

Floating point representation

In binary scientific notation, we write $x = \pm b_0.\,b_1 b_2 \cdots \times 2^m$, where $b_0, b_1, \ldots$ are all 0 or 1, with $b_0 = 1$ unless $x = 0$. The sequence $b_0.\,b_1 b_2 \cdots$ is called the mantissa and m the exponent.

In *double precision* 1 bit is used for the sign, 52 bits for the mantissa, and 11 bits for the exponent. The biased scheme is used to represent the exponent, which thus takes on values from -1023 to 1024. For the mantissa, 52 bits are used for $b_1, \ldots, b_{51}$ while the value of b_0 depends on m:

- If $m = -1023$ then $b_0 = 0$, which allows us to represent 0, using $b_1 = \cdots = b_{51} = 0$, or numbers smaller in size than 2^{-1023} otherwise (these are called denormalised numbers).
- If $-1023 < m < 1024$ then $b_0 = 1$.
- If $m = 1024$ then we use $b_1 = \cdots = b_{51} = 0$ to represent $\pm\infty$, which R writes as `-Inf` and `+Inf`. If one of the $b_i \neq 0$ then we interpret the representation as `NaN`, which stands for Not a Number.

In double precision, the smallest non-zero positive number is 2^{-1074} and the largest number is $2^{1023}(2-2^{-52})$ (sometimes called realmax).

When arithmetic operations on double precision floating point numbers produce a result smaller in magnitude than 2^{-1074} or larger in magnitude than realmax, then the result is 0 or $\pm\infty$, respectively. We call this *underflow* or *overflow*.

```
2^-1074 == 0
```

```
[1] FALSE
```

```
1/(2^-1074)
```

```
[1] Inf
```

```
2^1023+2^1022+2^1021
```

```
[1] 1.572981e+308
```

```
2^1023+2^1022+2^1022
```

The smallest number x such that $1 + x$ can be distinguished from 1 is $2^{-52} \approx 2.220446 \times 10^{-16}$, which is called *machine epsilon*.

Thus, in base 10, double precision is roughly equivalent to 16 significant figures, with exponents of size up to ± 308.

```r
x <- 1 + 2^-52
x - 1
```

```
[1] 2.220446e-16
```

```r
y <- 1 + 2^-53
y - 1
```

```
[1] 0
```

Significant digits

Using double precision numbers is roughly equivalent to working with 16 significant digits in base 10.

Arithmetic with integers will be exact for values from $-(2^{53}-1)$ to $2^{53}-1$ (roughly -10^{16} to 10^{16}), but as soon as you start using numbers outside this range, or fractions, you can expect to lose some accuracy due to *roundoff error*.

For example, 1.1 does not have a finite binary expansion, so in double precision its binary expansion is rounded to $1.00011001100\cdots001$, with an error of roughly 2^{-53}.

Absolute and Relative Error

- Let $\tilde{a}$ be an approximation of a, then the *absolute error* is $|\tilde{a} - a|$ and the *relative error* is $|\tilde{a} - a|/a$. Restricting $\tilde{a}$ to 16 significant digits is equivalent to allowing a relative error of 10^{-16}.

- When adding two approximations we add the absolute errors to get (a bound on) the absolute error of the result.

- When multiplying two approximations we add the relative errors to get (an approximation of) the relative error of result: suppose ε and δ are the (small) relative errors of a and b, then

$$\tilde{a}\tilde{b} = a(1+\varepsilon)b(1+\delta) = ab(1+\varepsilon+\delta+\varepsilon\delta) \approx ab(1+\varepsilon+\delta).$$

Catastrophic Cancellation

Suppose we add 1,000 numbers each of size around 1,000,000 with relative errors of up to 10^{-16}. Each thus has an absolute error of up to 10^{-10}, so adding them all we would have a number of size around 1,000,000,000 with an absolute error of up to 10^{-7}. That is, the relative error remains much the same at 10^{-16}.

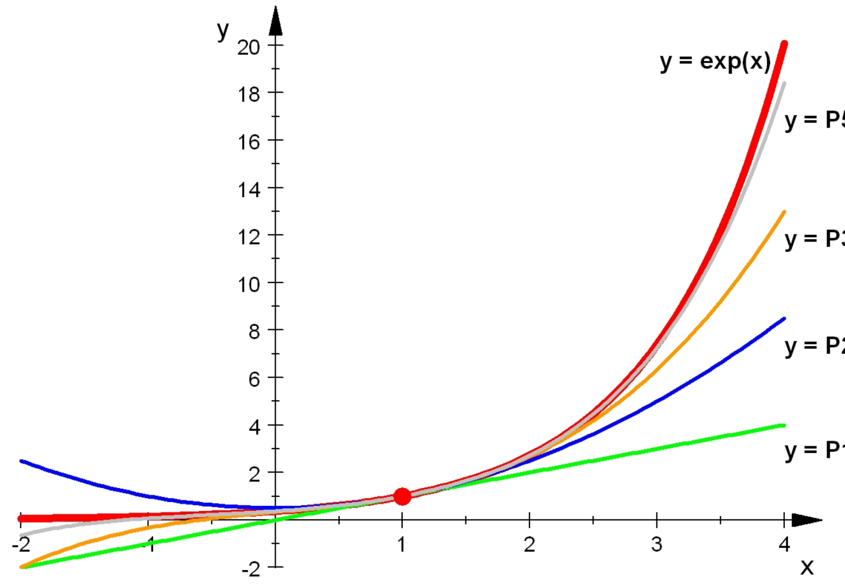
However, things can look very different when you start subtracting numbers of a similar size. For example, consider

$$1,234,567,812,345,678 - 1,234,567,800,000,000 = 12,345,678.$$

If the two numbers on the left-hand side have relative errors of 10^{-16}, then the right-hand side has an absolute error of about 1, which is a relative error of around 10^{-8}: a dramatic loss in accuracy, which we call *catastrophic cancellation* error.

Taylor series (approximation of $f(x)$ in the point a when $f(x)$ is infinitely differentiable):

$$
\begin{aligned}
f(x) &= f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots \\
&= \sum_{i=0}^{\infty} \frac{f^{(i)}(a)}{i!}(x-a)^i
\end{aligned}
$$



Taylor's Theorem

Let $f(x)$ be a $(n+1)$-times differentiable function. Then there exists a point ξ between a and x such that

$$\begin{aligned} f(x) &= P_n(x) + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1} \\ &\equiv P_n(x) + R_n(x) \end{aligned}$$

where

$$P_n(x) = \sum_i^n \frac{f^{(i)}(a)}{i!}(x-a)^i$$

denotes the n^{th}-order Taylor polynomial and $R_n(x)$ is the remainder term.

McLaurin series

Tayler series when $a = 0$

Example 1:

$$\begin{aligned} f(x) = e^x \rightarrow f'(x) &= e^x,\ f''(x) &= e^x,\ f^{(i)}(x) &= e^x \\ f(a) \quad = 1, \quad f'(a) &= 1, \quad f''(a) &= 1, \quad f^{(i)}(a) &= 1 \end{aligned}$$

$$\begin{aligned} f(x) &= f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2!} + f'''(a)\frac{(x-a)^3}{3!} + \dots \\ e^x &= 1 \quad + x \qquad\qquad\qquad + \frac{x^2}{2!} \qquad\qquad + \frac{x^3}{3!} \qquad\qquad + \dots \end{aligned}$$

McLaurin series: 2nd Example $f(x) = sin(x)$

$$\begin{aligned}
f'(x) &= cos(x) &\rightarrow f'(0) &= 1 \\
f''(x) &= -sin(x) &\rightarrow f''(0) &= 1 \\
f'''(x) &= -cos(x) &\rightarrow f'''(0) &= 1 \\
f''''(x) &= sin(x) &\rightarrow f''''(0) &= 1
\end{aligned}$$

This implies $f^{(2i)}(0) = 0$ and $f^{(2i+1)}(0) = (-1)^i$

$$\begin{aligned}
f(x) &= \sum_{i=0}^{\infty} f^{(i)}(a)\frac{(x-a)^i}{i!} \\
&= \sum_{i=0}^{\infty} f^{(2i)}(0)\frac{x^{2i}}{(2i)!} + \sum_{i=0}^{\infty} f^{(2i+1)}(0)\frac{x^{2i+1}}{(2i+1)!} = \\
&= 0 + \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{(2i+1)!}
\end{aligned}$$

Taylor series approximation (so with a finite number of terms):

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + O(x^{n+1})$$

Big O notation:

$g(x) = O(x^p)$ if for $x \to \infty$, $\frac{|g(x)|}{|x|^p} \leq M$ (with $M \in \mathbb{R}^+$ and $0 < |x| < r \in \mathbb{R}^+$)

Example: $\sin(x) - x$ near 0

If we wish to know $\sin(x) - x$ near 0, then we expect catastrophic cancellation to reduce the accuracy of our calculation. However

$$\sin(x) - x = \sum_{n=1}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

If we truncate this expansion to N terms, then the error is at most $|x^{2N+1}/(2N+1)!|$. Thus, using two terms

$$\sin(x) - x \approx -\frac{x^3}{6} + \frac{x^5}{120} = -\frac{x^3}{6}\left(1 - \frac{x^2}{20}\right).$$

If $|x| < 0.001$ then the error in the approximation is less than $0.001^5/120 < 10^{-17}$ in magnitude. If $|x| < 0.000001$ then the error is less than 10^{-302}. Since this formula does not involve subtraction of similarly sized numbers, it does not suffer from catastrophic cancellation.

Time

Ultimately we measure how efficient a program is by how long it takes to run. To measure how many CPU (Computer Processing Unit) seconds are spent evaluating an expression, we use `system.time(expression)`.

```
> system.time(source("scripts/prime.R"))

user  system elapsed
0.08    0.03    0.19
```

The sum of the user and system time gives the CPU seconds spent evaluating the expression.

The elapsed time also includes time spent on tasks unrelated to your R session.

- The time taken to sort a vector will depend on the length of the vector, n.
- The time taken to solve the equation $f(x) = 0$, accurate to within some tolerance ε, will depend on ε.
- We assess efficiency by counting the number of operations executed in running a program, where operations are tasks such as addition, multiplication, logical comparison, variable assignment, and calling built-in functions.

For example, the following program will sum the elements of a vector `x`:

```
S <- 0
for (a in x) S <- S + a
```

Let n be the length of `x`, then when we run this program we carry out n addition operations and $2n + 1$ variable assignments (we assign a value to `a` and to `S` each time we go around the `for` loop).

For a second example, suppose we are using the Taylor series $\sum_{n=1}^{N}(-1)^{n+1}x^n/n$ to approximate $\log(1+x)$, for $0 \le x \le 1$, and we want an error of at most $\pm\varepsilon$.

It can be shown that the approximation error is no greater in magnitude than the last term in the sum, which suggests the following code:

```
eps <- 1e-12
x <- 0.5

n <- 0
log1x <- 0
while (n == 0 || abs(last.term) > eps) {
  n <- n + 1
  last.term <- (-1)^(n+1)*x^n/n
  log1x <- log1x + last.term
}
```

When we go around the loop the n-th time we perform three additions and $2n+3$ multiplications/divisions.

We loop until $x^n/n < \varepsilon$. For $x \in (0,1]$ we get $n \leq \lceil 1/\varepsilon \rceil$.

Thus the total number of additions will be bounded by $3\lceil 1/\varepsilon \rceil$ and the total number of multiplications/divisions bounded by

$$\sum_{n=1}^{\lceil 1/\varepsilon \rceil} (2n+3) = \lceil 1/\varepsilon \rceil (\lceil 1/\varepsilon \rceil + 1) + 3\lceil 1/\varepsilon \rceil = \lceil 1/\varepsilon \rceil^2 + 4\lceil 1/\varepsilon \rceil.$$

In this example, a simple modification to the program will improve the efficiency.

```
eps <- 1e-12
x <- 0.5

log1x <- x
last.term <- x
n <- 1
while (abs(last.term) > eps) {
  n <- n + 1
  last.term <- -last.term*x*(1 - 1/n)
  log1x <- log1x + last.term
}
```

We now have just three multiplications/divisions each time we go around the loop, so the total number will be bounded by $3\lceil 1/\varepsilon \rceil$.

- If we know the number of operations grows like an^b where n is the problem size (the length of the vector or the inverse tolerance in our examples), then the value of b is *much* more important than the value of a.

- Let f and g be functions of n, then we say that
  - $f(n) = O(g(n))$ as $n \to \infty$ if $\lim_{n\to\infty} f(n)/g(n) < \infty$, and
  - $f(n) = o(g(n))$ as $n \to \infty$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.

- Our first example required $O(n)$ operations to sum a vector of length n. In its initial form, our second example required $O(1/\varepsilon^2)$ operations to calculate $\log(1+x)$ to within tolerance ε.

- Some operations take much longer than others: Transcendental functions such as $\sin$ and $\log$ take more time than simple addition or substraction.

- In practice what we do is identify the longest or most important operation in a program, and count how many times it is performed.

- For example, for numerical integration, root-finding, and optimisation, we are working with a user-defined function f, and we count how many times $f(x)$ is evaluated, for different x. For numerical sorting algorithms, we count how many comparisons of the form $x < y$ are made.

Creating or changing the size of a vector (also called redimensioning an array) is relatively slow, which is why, when we know how big a vector is going to be, it is better to initialise it fully grown (but full of zeros) than to increase it incrementally. We can compare the relative speeds using `system.time`:

```
> n<-1e8
> x<-rep(0,n)
> system.time(for (i in 1:n) x[i] <- i^2)

   user  system elapsed
   4.70    0.00    4.71

> x <- c()
> system.time(for (i in 1:n) x[i] <- i^2)

   user  system elapsed
  21.28    1.12   22.41
```

Loops versus vectors

- In R, vector operations are generally faster than equivalent loops.
- When you evaluate an expression in R, it is "translated" into a faster lower-level language before being evaluated, then the result is translated back into R.
- For example, take the following code to square each element of x:

```r
for (i in 1:length(x)) x[i] <- x[i]^2
```

- Each time we evaluate the expression `x[i] <- x[i]^2`, we have to translate `x[i]` into our lower-level language, and then translate the result back.
- In contrast, to evaluate the expression `x <- x^2`, we translate `x` all at once and then square it, before translating the answer back: all the work takes place in our faster lower-level language.

Example: column sums of a matrix

```
> big.matrix <- matrix(1:1e8, nrow=1e4)
> colsums <- rep(NA, dim(big.matrix)[2])
> system.time({
  for (i in 1:dim(big.matrix)[2]) {
    s <- 0
    for (j in 1:dim(big.matrix)[1]) {
      s <- s + big.matrix[j,i]
    }
    colsums[i] <- s
  }
})

   user  system elapsed
   4.48    0.00    4.49
```

```
> system.time(colsums <- apply(big.matrix, 2, sum))

   user  system elapsed
   1.00    0.11    1.11

> system.time(
   for (i in 1:dim(big.matrix)[2]) {
     colsums[i] <- sum(big.matrix[,i])
   }
 )

   user  system elapsed
   0.16    0.09    0.25

> system.time(colsums <- colSums(big.matrix))

   user  system elapsed
   0.11    0.00    0.10
```

Memory

- Computer memory comes in a variety of forms.
- Variables require memory.
- Because accessing an existing variable is invariably quicker than recalculating it, it is usual to store commonly used quantities for reuse.

For example, consider the function `prime`

```r
prime <- function(n) {
    # returns TRUE if n is prime
    # assumes n is a positive integer
    if (n == 1) {
        is.prime <- FALSE
    } else if (n == 2) {
        is.prime <- TRUE
    } else {
        is.prime <- TRUE
        m <- 2
        m.max <- sqrt(n)  # only want to calculate this once
        while (is.prime && m <= m.max) {
            if (n %% m == 0) is.prime <- FALSE
            m <- m + 1
        }
    }
    return(is.prime)
}
```

- Calculating $\sqrt{n}$ is relatively slow, so we do this once and store the result.
- An alternative is for the main while loop to start as follows:

```
while (is.prime && m <= sqrt(n))
```

- Coding the loop this way would require us to recalculate $\sqrt{n}$ each time we check the loop condition, which is inefficient.

Summary

- Because R works much more quickly with vectors than loops, it is usual to try to vectorise R programs.
- If a vector (or list) is too large to store in RAM all at once, then it may not be possible to store it at all, in which case you are said to have run out of memory.
- R has an absolute limit on the length of a vector of $2^{31} - 1 =$ 2,147,483,647
- If you find this happening then you will need to break your vectors down into smaller subvectors and deal with each in turn.
- In extreme cases it may be necessary to save a variable and then delete it from the workspace, using `save` and `rm`.