
Robin J. Evans
www.stats.ox.ac.uk/~evans

Michaelmas 2014
Algorithmic Considerations

Some methods of computing things are easier than others.

Multiplying $n \times m$ matrix by $m \times k$ matrix is $O(nmk)$ calculations.
Algorithmic Considerations

Some methods of computing things are easier than others.

Multiplying $n \times m$ matrix by $m \times k$ matrix is $O(nmk)$ calculations.

Let $A$ be $n \times m$, $B$ be $m \times k$, and $c$ be $k \times 1$.

$$ABc = (AB)c = A(Bc)$$

But $(AB)c$ takes $O(nmk + nk)$, and $A(Bc)$ takes $O(mk + nm)$. 

\[
\begin{align*}
A &= \text{matrix(rnorm(1e4), 100, 100); } \\
B &= \text{matrix(rnorm(1e4), 100, 100); } \\
c &= \text{rnorm(100); } \\
\text{library(microbenchmark); } \\
\text{microbenchmark(A %*% B %*% c, A %*% (B %*% c), times=100; } \\
\text{## Unit: microseconds; } \\
\text{## expr min lq median uq max neval; } \\
\text{## A %*% B %*% c 700.8 711.96 715.41 719.91 1426.40 100; } \\
\text{## A %*% (B %*% c) 64.8 65.11 65.22 65.52 85.81 100; } \\
\text{R evaluates left to right in this case.}
\end{align*}
\]
Algorithmic Considerations

Some methods of computing things are easier than others.

Multiplying $n \times m$ matrix by $m \times k$ matrix is $O(nmk)$ calculations.

Let $A$ be $n \times m$, $B$ be $m \times k$, and $c$ be $k \times 1$.

$$ABc = (AB)c = A(Bc)$$

But $(AB)c$ takes $O(nmk + nk)$, and $A(Bc)$ takes $O(mk + nm)$.

```r
A = matrix(rnorm(1e4), 100, 100); B = matrix(rnorm(1e4), 100, 100)
c = rnorm(100)

library(microbenchmark)
microbenchmark(A %*% B %*% c, A %*% (B %*% c), times=100)
```

## Unit: microseconds

<table>
<thead>
<tr>
<th>expr</th>
<th>min</th>
<th>lq</th>
<th>median</th>
<th>uq</th>
<th>max</th>
<th>neval</th>
</tr>
</thead>
<tbody>
<tr>
<td>A %<em>% B %</em>% c</td>
<td>700.8</td>
<td>711.96</td>
<td>715.41</td>
<td>719.91</td>
<td>1426.40</td>
<td>100</td>
</tr>
<tr>
<td>A %<em>% (B %</em>% c)</td>
<td>64.8</td>
<td>65.11</td>
<td>65.22</td>
<td>65.52</td>
<td>85.81</td>
<td>100</td>
</tr>
</tbody>
</table>

R evaluates left to right in this case.
Numerical Stability Considerations

Floating point numbers have a limited accuracy (usually around $10^{-16}$ for an $O(1)$ number).

\[ 0.3 - 0.2 - 0.1 \]

## [1] -2.776e-17
Numerical Stability Considerations

Floating point numbers have a limited accuracy (usually around $10^{-16}$ for an $O(1)$ number).

\[ 0.3 - 0.2 - 0.1 \]

```
## [1] -2.776e-17
```

```r
summary(A %*% B %*% c - A %*% (B %*% c))
```

```
##          V1
##  Min.    : -1.99e-13
##  1st Qu. : -2.84e-14
##  Median  :  1.42e-14
##  Mean    :  8.52e-15
##  3rd Qu. :  4.35e-14
##  Max.    :  2.42e-13
```
Numerical Stability Considerations

Floating point numbers have a limited accuracy (usually around $10^{-16}$ for an $O(1)$ number).

0.3 - 0.2 - 0.1

## [1] -2.776e-17

summary(A %*% B %*% c - A %*% (B %*% c))

## V1
## Min. : -1.99e-13
## 1st Qu.: -2.84e-14
## Median : 1.42e-14
## Mean : 8.52e-15
## 3rd Qu.: 4.35e-14
## Max. : 2.42e-13

Problems of numerical accuracy can be solved with long doubles in languages like C.
Arithmetic Precision
R may hide some of these rounding issues, so don’t forget that they exist!

```r
1 + 1e-15
## [1] 1
print(1 + 1e-15, digits = 22)
## [1] 1.000000000000001110223
```

R also has an integer type (but it’s a bit tricky)

```r
1 == 1L
## [1] TRUE
identical(1, 1L)
## [1] FALSE
```
Arithmetic Precision

R may hide some of these rounding issues, so don’t forget that they exist!

```r
1+1e-15

## [1] 1

print(1+1e-15, digits=22)

## [1] 1.000000000000001110223
```

R also has an integer type (but it’s a bit tricky)

```r
1 == 1L

## [1] TRUE

identical(1, 1L)

## [1] FALSE
```
Equality testing may be problematic with floating point numbers:

```r
x = 1+1e-15
x == 1

## [1] FALSE

all.equal(x, 1)

## [1] TRUE
```
Arithmetic Precision

Equality testing may be problematic with floating point numbers:

```
x = 1+1e-15
x == 1
```

```# [1] FALSE```

```
all.equal(x, 1)
```

```# [1] TRUE```

`all.equal()` ignores small differences in numbers, and also their type.

```
all.equal(1L, 1)
```

```# [1] TRUE```
Numerical Stability Considerations

Floats also have an upper and lower limits on the numbers they can hold

\[
\begin{align*}
\text{c}(2^{-1074}, \ 2^{-1075}) \\
\text{## [1]} & \quad 4.941\text{e}-324 \quad \text{0.000e+00} \\
\text{c}(2^{1023}, \ 2^{1024}) \\
\text{## [1]} & \quad 8.988\text{e}+307 \quad \text{Inf}
\end{align*}
\]
Numerical Stability Considerations

Floats also have an upper and lower limits on the numbers they can hold

```r
c(2^-1074, 2^-1075)
```

```r
## [1] 4.941e-324 0.000e+00
```

```r
c(2^1023, 2^1024)
```

```r
## [1] 8.988e+307 Inf
```

You may need to think carefully about the way in which you compute things

```r
```

```r
## [1] 128 NaN
```
Numerical Stability Considerations

Additive computations are generally much more stable than multiplicative ones.
Numerical Stability Considerations

Additive computations are generally much more stable than multiplicative ones. Suppose you want to calculate the geometric mean of some numbers.

```r
set.seed(324)
geomean = function(x) prod(x)^(1/length(x))
x = rlnorm(1e3, meanlog=-1)  # log-normals
geomean(x)

## [1] 0

range(x)

## [1] 0.01134 9.73558
```
Numerical Stability Considerations

Additive computations are generally much more stable than multiplicative ones. Suppose you want to calculate the geometric mean of some numbers.

```r
set.seed(324)
geomean = function(x) prod(x)^(1/length(x))
x = rlnorm(1e3, meanlog=-1)  # log-normals
geomean(x)
```

```r
## [1] 0
```

```r
range(x)
```

```r
## [1] 0.01134 9.73558
```

If we do everything on a log-scale, there’s no problem.

```r
geomean2 = function(x) exp(mean(log(x)))
geomean2(x)  # approximately exp(-1)
```

```r
## [1] 0.3597
```