LECTURE 13: NUMERICAL ISSUES IN R

STAT 598z: Introduction to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

February 23, 2017

Binary representation

Most computation ignores that computers approximate math

Or assume (hope?) that approximations never get too bad

Binary representation

Most computation ignores that computers approximate math

Or assume (hope?) that approximations never get too bad

'Numerical Issues in Stat. Computing for the Social Scientist':

- Rerunning analysis on modern computers produce much weaker link between pollution and health problems
- Rockets/space probes crash because of numerical issues
- False 'discoveries' in physics

Integers

Integers in `R` are currently stored using 32-bits
(irrespective of your machine hardware, but see `bit64` package)

$$SB_1B_2\cdots B_{31}$$

The first bit `S` is the sign-bit

Integers

Integers in `R` are currently stored using 32-bits
(irrespective of your machine hardware, but see `bit64` package)

$$\mathrm{S}\mathrm{B}_1\mathrm{B}_2\cdots\mathrm{B}_{31}$$

The first bit `S` is the sign-bit

`R` can *exactly* represent integers from $-(2^{31}-1)$ to $(2^{31}-1)$

```
as.integer(2^30)
as.integer(2^31)
as.integer(2^30) + (as.integer(2^30) - 1L)  # Note the order
```

Integers

Integers in `R` are currently stored using 32-bits
(irrespective of your machine hardware, but see `bit64` package)

$$SB_1B_2\cdots B_{31}$$

The first bit `S` is the sign-bit

`R` can *exactly* represent integers from $-(2^{31}-1)$ to $(2^{31}-1)$

```
as.integer(2^30)
as.integer(2^31)
as.integer(2^30) + (as.integer(2^30) - 1L)  # Note the order
```

Organize your computations to avoid overflow

Integers

Integers in `R` are currently stored using 32-bits
(irrespective of your machine hardware, but see `bit64` package)

$$\mathrm{SB}_1\mathrm{B}_2\cdots\mathrm{B}_{31}$$

The first bit `S` is the sign-bit

`R` can *exactly* represent integers from $-(2^{31}-1)$ to $(2^{31}-1)$

```
as.integer(2^30)
as.integer(2^31)
as.integer(2^30) + (as.integer(2^30) - 1L)  # Note the order
```

Organize your computations to avoid overflow

Integers are useful for indexing vectors or interfacing with `C`

Doubles

Doubles: double-precision floating point numbers

Stored using 64-bits in the IEEE 754 format.

Doubles

Doubles: double-precision floating point numbers

Stored using 64-bits in the IEEE 754 format. This includes:

- representation of finite numbers, infinities, NaN's
- rounding rules
- exception handling (e.g. division by zero)
- operations (like arithmetic)

Doubles

Doubles: double-precision floating point numbers

Stored using 64-bits in the IEEE 754 format. This includes:

- representation of finite numbers, infinities, NaN's
- rounding rules
- exception handling (e.g. division by zero)
- operations (like arithmetic)

Reals are rounded to the nearest floating point numbers

Floating points arithmetic: a leaky abstraction

`http://www.johndcook.com/blog/2009/04/06/numbers-are-a-leaky-abstraction/`

$$\underbrace{\mathtt{S}}_{\texttt{Sign}}\ \underbrace{\mathtt{M}_1\mathtt{M}_2\cdots\mathtt{M}_{52}}_{\texttt{Mantissa}}\ \underbrace{\mathtt{E}_1\mathtt{E}_2\cdots\mathtt{E}_{11}}_{\texttt{Exponent}}$$

Corresponds (usually) to the value

- $(-1)^{\mathsf{S}} \times 1.\mathsf{M} \times 2^{\mathsf{E}-\mathsf{e}}$

Floating point numbers

$$\underbrace{\mathtt{S}}_{\mathtt{Sign}}\ \underbrace{\mathtt{M}_1\mathtt{M}_2\cdots\mathtt{M}_{52}}_{\mathtt{Mantissa}}\ \underbrace{\mathtt{E}_1\mathtt{E}_2\cdots\mathtt{E}_{11}}_{\mathtt{Exponent}}$$

Corresponds (usually) to the value

- $(-1)^{\mathtt{S}} \times 1.\mathtt{M} \times 2^{\mathtt{E}-\mathtt{e}}$ i.e.
- $(-1)^{\mathtt{S}} \times \left(1 + \left(\sum_{\mathtt{p}=1}^{52} \mathtt{M}_{\mathtt{p}} 2^{-\mathtt{p}}\right)\right) 2^{\mathtt{E}-\mathtt{e}}$, where $\mathtt{E} = \sum_{\mathtt{i}=1}^{11} \mathtt{E}_{12-\mathtt{i}} 2^{\mathtt{i}-\mathtt{e}}$

$\mathtt{e} = 1023$ is the bias of the exponent

$$\underbrace{\mathtt{S}}_{\texttt{Sign}}\ \underbrace{\mathtt{M_1 M_2 \cdots M_{52}}}_{\texttt{Mantissa}}\ \underbrace{\mathtt{E_1 E_2 \cdots E_{11}}}_{\texttt{Exponent}}$$

Corresponds (usually) to the value

- $(-1)^{\mathtt{S}} \times \mathtt{1.M} \times 2^{\mathtt{E}-\mathtt{e}}$ i.e.
- $(-1)^{\mathtt{S}} \times \left(1 + \left(\sum_{p=1}^{52} \mathtt{M}_p 2^{-p}\right)\right) 2^{\mathtt{E}-\mathtt{e}}$, where $\mathtt{E} = \sum_{i=1}^{11} \mathtt{E}_{12-i} 2^{i-e}$

$\mathtt{e} = 1023$ is the bias of the exponent

All leading/trailing zeros are removed by adjusting exponent
Called Normalized form: decimal point after first significant bit

See `http://www.h-schmidt.net/FloatConverter/IEEE754.html`

Floating point numbers

$$\underbrace{\mathtt{S}}_{\texttt{Sign}}\ \underbrace{\mathtt{M}_1\mathtt{M}_2\cdots\mathtt{M}_{52}}_{\texttt{Mantissa}}\ \underbrace{\mathtt{E}_1\mathtt{E}_2\cdots\mathtt{E}_{11}}_{\texttt{Exponent}}$$

Floating point numbers

$$\underbrace{\mathtt{S}}_{\mathtt{Sign}}\ \underbrace{\mathtt{M}_1\mathtt{M}_2\cdots\mathtt{M}_{52}}_{\mathtt{Mantissa}}\ \underbrace{\mathtt{E}_1\mathtt{E}_2\cdots\mathtt{E}_{11}}_{\mathtt{Exponent}}$$

Most real numbers must be rounded to nearest approximation

E.g. 0.2 has no finite (binary) floating-point representation

$$\underbrace{\mathtt{S}}_{\mathtt{Sign}}\ \underbrace{\mathtt{M}_1\mathtt{M}_2\cdots\mathtt{M}_{52}}_{\mathtt{Mantissa}}\ \underbrace{\mathtt{E}_1\mathtt{E}_2\cdots\mathtt{E}_{11}}_{\mathtt{Exponent}}$$

Most real numbers must be rounded to nearest approximation

E.g. 0.2 has no finite (binary) floating-point representation

A number between 1 and 2 has about $2^{-52} = 10^{-16}$ accuracy

```
0.3-0.2-0.1
```

Floating point numbers

$$\underbrace{\mathsf{S}}_{\text{Sign}}\ \underbrace{\mathsf{M}_1\mathsf{M}_2\cdots\mathsf{M}_{52}}_{\text{Mantissa}}\ \underbrace{\mathsf{E}_1\mathsf{E}_2\cdots\mathsf{E}_{11}}_{\text{Exponent}}$$

Most real numbers must be rounded to nearest approximation

E.g. 0.2 has no finite (binary) floating-point representation

A number between 1 and 2 has about $2^{-52} = 10^{-16}$ accuracy

```
0.3-0.2-0.1
```

R can hide internal warts from you

```
print(0.3*4)
print(0.3*4, digits=20)
```

$$\underbrace{\mathtt{S}}_{\text{Sign}}\ \underbrace{\mathtt{M_1M_2\cdots M_{52}}}_{\text{Mantissa}}\ \underbrace{\mathtt{E_1E_2\cdots E_{11}}}_{\text{Exponent}}$$

We saw that `1.2 == 0.4*3` is FALSE

Instead, can use `all.equal()` (has a tolerance of about 10^{-8})

```
> isTRUE(all.equal(1, 1 + 1e-8))
[1] TRUE
> isTRUE(all.equal(1, 1 + 2e-8))
[1] FALSE
```

For details, see http://stackoverflow.com/questions/15334701/how-does-the-tolerance-parameter-of-all-equal-work

Floating point numbers

$$\underbrace{\mathtt{S}}_{\mathtt{Sign}}\ \underbrace{\mathtt{M_1M_2\cdots M_{52}}}_{\mathtt{Mantissa}}\ \underbrace{\mathtt{E_1E_2\cdots E_{11}}}_{\mathtt{Exponent}}$$

We saw that `1.2 == 0.4*3` is `FALSE`

Instead, can use `all.equal()` (has a tolerance of about 10^{-8})

```
> isTRUE(all.equal(1, 1 + 1e-8))
[1] TRUE
> isTRUE(all.equal(1, 1 + 2e-8))
[1] FALSE
```

For details, see `http://stackoverflow.com/questions/15334701/how-does-the-tolerance-parameter-of-all-equal-work`

Also useful is the `near()` function from package `dplyr`

```
> near(.1+.2,.3)
```

Finite storage also leads to overflow/underflow

11 bits in exponent allows range of about ±`1e-308` to ±`1e308`

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

Overflow and underflow

Finite storage also leads to overflow/underflow

11 bits in exponent allows range of about $\pm$`1e-308` to $\pm$`1e308`

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

Organize computations to avoid intermediate over/underflow

```
c( 2^2000/2^1990, 2^(2000 - 1990) )
```

`http://www.stats.ox.ac.uk/~evans/CDT/Slides.pdf`

Inverting matrices

Consider a system of linear equations:

$$y = \mathsf{A}x$$

Given $y \in \mathbb{R}^n$ and $\mathsf{A} \in \mathbb{R}^{n \times n}$, we want to find x:

$$x = \mathsf{A}^{-1}y$$

Inverting matrices

Consider a system of linear equations:

$$y = \mathbf{A}x$$

Given $y \in \mathbb{R}^n$ and $\mathbf{A} \in \mathbb{R}^{n \times n}$, we want to find x:

$$x = \mathbf{A}^{-1}y$$

In R:

```
x <- solve(A) %*% y    # Inefficient and numerically inaccurate
```

Inverting matrices

Consider a system of linear equations:

$$y = \mathsf{A}x$$

Given $y \in \mathbb{R}^n$ and $\mathsf{A} \in \mathbb{R}^{n \times n}$, we want to find x:

$$x = \mathsf{A}^{-1}y$$

In R:

```
x <- solve(A) %*% y    # Inefficient and numerically inaccurate
```

Much better:

```
x <- solve(A,y)
```

Other functions to avoid numerical issues

`log1p(x)`: calculates $\log(1 + x)$

- What if x is very small? $1 + x$ has less precision than x!

Other functions to avoid numerical issues

`log1p(x)`: calculates $\log(1 + x)$

- What if x is very small? $1 + x$ has less precision than x!

`expm1(x)`: calculates $\exp(x) - 1$

- What if x is very small?

Other functions to avoid numerical issues

`log1p(x)`: calculates $\log(1 + x)$

- What if x is very small? $1 + x$ has less precision than x!

`expm1(x)`: calculates $\exp(x) - 1$

- What if x is very small?

`erfc(x)`: calculates $1 - \text{erf}(x)$

- Floating point represents small deviations from 0 more precisely than from 1

Other functions to avoid numerical issues

`log1p(x)`: calculates $\log(1 + x)$

- What if x is very small? $1 + x$ has less precision than x!

`expm1(x)`: calculates $\exp(x) - 1$

- What if x is very small?

`erfc(x)`: calculates $1 - \mathrm{erf}(x)$

- Floating point represents small deviations from 0 more precisely than from 1

`lfactorial(x)`, and `lgamma(x)`: calculate log factorial/Gamma

Other functions to avoid numerical issues

`log1p(x)`: calculates $\log(1 + x)$

- What if x is very small? $1 + x$ has less precision than x!

`expm1(x)`: calculates $\exp(x) - 1$

- What if x is very small?

`erfc(x)`: calculates $1 - \mathrm{erf}(x)$

- Floating point represents small deviations from 0 more precisely than from 1

`lfactorial(x)`, and `lgamma(x)`: calculate log factorial/Gamma

`http://www.johndcook.com/blog/2010/06/07/math-library-functions-that-seem-unnecessary/`

Storing log-probabilities

Storing numbers as logarithms allows larger dynamic range

`log` also convert multiplications to addition

Storing log-probabilities

Storing numbers as logarithms allows larger dynamic range

`log` also convert multiplications to addition

For a set $X = \{x_1, \cdots, x_N\}$ of independent observations:

$$p(X) = \prod_{i=1}^{N} p(x_i)$$

Storing log-probabilities

Storing numbers as logarithms allows larger dynamic range

`log` also convert multiplications to addition

For a set $X = \{x_1, \cdots, x_N\}$ of independent observations:

$$p(X) = \prod_{i=1}^{N} p(x_i)$$

E.g. a (small) set of 1000 standard normals:

```
> prod(pnorm(rnorm(1000)))
[1] 0
```

Storing log-probabilities

Storing numbers as logarithms allows larger dynamic range

`log` also convert multiplications to addition

For a set $X = \{x_1, \cdots, x_N\}$ of independent observations:

$$p(X) = \prod_{i=1}^{N} p(x_i)$$

E.g. a (small) set of 1000 standard normals:

```
> prod(pnorm(rnorm(1000)))
[1] 0
> sum(log(pnorm(rnorm(1000))))        # Better
[1] -1032.019
```

Storing log-probabilities

Storing numbers as logarithms allows larger dynamic range

`log` also convert multiplications to addition

For a set $X = \{x_1, \cdots, x_N\}$ of independent observations:

$$p(X) = \prod_{i=1}^{N} p(x_i)$$

E.g. a (small) set of 1000 standard normals:

```
> prod(pnorm(rnorm(1000)))
[1] 0
> sum(log(pnorm(rnorm(1000))))        # Better
[1] -1032.019
> sum(pnorm(rnorm(1000),log.p=TRUE))  # Even better!
[1] -1032.019
```

Calculating a Gaussian probability

Recall, with mean μ and covariance Σ,

$$p(x) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

Calculating a Gaussian probability

Recall, with mean μ and covariance Σ,

$$p(x) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

$$\log p(x) = -\log(2\pi) - \log|\Sigma| - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)$$

Calculating a Gaussian probability

Recall, with mean μ and covariance Σ,

$$p(x) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

$$\log p(x) = -\log(2\pi) - \log|\Sigma| - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)$$

```
p <- log(2*pi) + determinant(sigma, log=T)[[1]] -
      0.5 * t(x-mu) %*% solve(sigma, (x-mu))
```

Calculating a Gaussian probability

Recall, with mean μ and covariance Σ,

$$p(x) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

$$\log p(x) = -\log(2\pi) - \log|\Sigma| - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)$$

```
p <- log(2*pi) + determinant(sigma, log=T)[[1]] -
      0.5 * t(x-mu) %*% solve(sigma, (x-mu))
```



```
p <- log(2*pi) + determinant(sigma, log=T)[[1]] -
      0.5 * t(x-mu) %*% solve(sigma, (x-mu))
```

Calculating a Gaussian probability

Recall, with mean μ and covariance Σ,

$$p(x) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

$$\log p(x) = -\log(2\pi) - \log|\Sigma| - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)$$

```
p <- log(2*pi) + determinant(sigma, log=T)[[1]] -
      0.5 * t(x-mu) %*% solve(sigma, (x-mu))
```


```
p <- log(2*pi) + determinant(sigma, log=T)[[1]] -
      0.5 * t(x-mu) %*% solve(sigma, (x-mu))
```


Determinant and inversion require Cholesky decomposition

Using `log` turns multiplication into addition

What if we wanted addition in the first place?

Log-sum-exp

Using `log` turns multiplication into addition

What if we wanted addition in the first place?

E.g. for two disjoint events A and B,

$$P(A \text{ or } B) = P(A) + P(B)$$

Log-sum-exp

Using `log` turns multiplication into addition

What if we wanted addition in the first place?

E.g. for two disjoint events A and B,

$$P(A \text{ or } B) = P(A) + P(B)$$

We only store $\log P(A)$ and $\log P(B)$

How do we calculate $\log(P(A) + P(B))$?

Log-sum-exp

Using `log` turns multiplication into addition

What if we wanted addition in the first place?

E.g. for two disjoint events A and B,

$$P(A \text{ or } B) = P(A) + P(B)$$

We only store $\log P(A)$ and $\log P(B)$

How do we calculate $\log(P(A) + P(B))$?

First exponentiate log-probs, add them and then take log?

Log-sum-exp

Using `log` turns multiplication into addition

What if we wanted addition in the first place?

E.g. for two disjoint events A and B,

$$P(A \text{ or } B) = P(A) + P(B)$$

We only store $\log P(A)$ and $\log P(B)$

How do we calculate $\log(P(A) + P(B))$?

First exponentiate log-probs, add them and then take log?

This operation, `logsumexp(P(A), P(B))`, leads to over/underflow

Log-sum-exp

Let $M = \max(P(A), P(B))$

$$P(A) + P(B) = M \times \frac{P(A) + P(B)}{M}$$

Log-sum-exp

Let $M = \max(P(A), P(B))$

$$P(A) + P(B) = M \times \frac{P(A) + P(B)}{M}$$
$$\log(P(A) + P(B)) = \log M + \log\left(\frac{P(A)}{M} + \frac{P(B)}{M}\right)$$

Log-sum-exp

Let $M = \max(P(A), P(B))$

$$P(A) + P(B) = M \times \frac{P(A) + P(B)}{M}$$
$$\log(P(A) + P(B)) = \log M + \log\left(\frac{P(A)}{M} + \frac{P(B)}{M}\right)$$

Now calculate `logsumexp` $\left(\frac{\texttt{P(A)}}{\texttt{M}}, \frac{\texttt{P(B)}}{\texttt{M}}\right)$
i.e. exponentiate, sum and take logarithm

Log-sum-exp

Let $M = \max(P(A), P(B))$

$$P(A) + P(B) = M \times \frac{P(A) + P(B)}{M}$$
$$\log(P(A) + P(B)) = \log M + \log\left(\frac{P(A)}{M} + \frac{P(B)}{M}\right)$$

Now calculate `logsumexp` $\left(\frac{\mathtt{P(A)}}{\mathtt{M}}, \frac{\mathtt{P(B)}}{\mathtt{M}}\right)$
i.e. exponentiate, sum and take logarithm

What have we gained?

- at least one of $\frac{P(A)}{M}, \frac{P(B)}{M}$ is nonzero (one actually)
- both are finite