Stat 3701 Lecture Notes: Computer Arithmetic

Charles J. Geyer

February 07, 2020

1 License

2 R

- The version of R used to make this document is 3.6.2.
- The version of the `rmarkdown` package used to make this document is 2.1.
- The version of the `numDeriv` package used to make this document is 2016.8.1.1.

3 IEEE Arithmetic

What is for short called IEEE arithmetic is a standard for floating point arithmetic implemented in nearly all currently manufactured computers.

What you need to know about IEEE arithmetic is that there are several kinds of floating point numbers. In C and C++ the types are

- `float` about 6.9 decimal digits precision,
- `double` about 15.6 decimal digits precision, and
- `long double` which can be anything, often the same as `double`.

In R only `double` is used.

IEEE arithmetic also represents values that are not ordinary floating point numbers. In R these are printed

- `NaN` "meaning" *not a number*,
- `Inf` "meaning" $+\infty$,
- `-Inf` "meaning" $-\infty$,

in all three cases the scare quotes around "meaning" mean the meaning is more complicated than first appears, as we shall see as we go along.

These follow obvious rules of arithmetic

$$
\begin{aligned}
\texttt{NaN} + x &= \texttt{NaN} \\
\texttt{NaN} * x &= \texttt{NaN} \\
\texttt{Inf} + x &= \texttt{Inf}, \qquad x > \texttt{-Inf} \\
\texttt{Inf} + \texttt{-Inf} &= \texttt{NaN} \\
\texttt{Inf} * x &= \texttt{Inf}, \qquad x > 0 \\
\texttt{Inf} * 0 &= \texttt{NaN} \\
x/0 &= \texttt{Inf}, \qquad x > 0 \\
0/0 &= \texttt{NaN}
\end{aligned}
$$

4 Overflow

In R the function `is.finite` tests that numbers are not any of `NA`, `NaN`, `Inf`, `-Inf`.

It can happen that `all(is.finite(x))` is `TRUE` but `sum(x)` or `prod(x)` is `Inf`. This is called overflow.

Overflow must be avoided if at all possible. It loses all significant figures.

Example:

```
log(exp(710))
```

```
## [1] Inf
```

5 Underflow

An IEEE arithmetic result can be zero, when the exact infinite-precision result would be positive but smaller than the smallest positive number representable in IEEE arithmetic. This is called underflow.

Example:

```
log(exp(-746))
```

```
## [1] -Inf
```

Underflow is not a worry if the result is later added to a large number.

Example:

```
1 + exp(-746)
```

```
## [1] 1
```

is very close to correct, as close to correct as the computer can represent.

6 Denormalized Numbers

Between the smallest positive number representable with full (15.6 decimal digit) precision and zero are numbers representable with less precision.

Example:

```
log(exp(-743))
```

```
## [1] -743.0538
```

Theoretically, since `log` and `exp` are inverses of each other, we should get -743 as the answer. But `exp(-743)` is a denormalized number with less than full precision, so we only get close but not very close to the correct result.

7 Catastrophic Cancellation

We say "catastrophic cancellation" occurs when subtracting two nearly equal positive numbers gives a number with much less precision.

Example

$$1.020567 - 1.020554 = 1.3 \times 10^{-5}$$

Both operands have 7 decimal digits of precision. The result has 2.

That's if we are assuming decimal arithmetic. Computers, of course, use binary arithmetic, but the principle is the same.

7.1 The Complement Rule is Wrong

What I call the "complement rule" is the simplest fact of probability theory

$$\Pr(\text{not } A) = 1 - \Pr(A), \qquad \text{for any event } A.$$

But it assumes *real* real numbers, not the computer's sorta-kinda real numbers (`doubles`).

The complement rule doesn't work in the upper tail of probability distributions where probabilities are nearly equal to one. R, being (unlike C and C++) a computer language highly concerned with numerical accuracy, provides a workaround. All of the "`p`" and "`q`" functions like `pnorm` and `qnorm` have a `lower.tail` argument to work around this issue.

```
pnorm(2.5, lower.tail = FALSE)
```

```
## [1] 0.006209665
```

```
1 - pnorm(2.5)
```

```
## [1] 0.006209665
```

Same thing, right? But the latter, shorter and simpler though it may seem, suffers from catastrophic cancellation.

```
x <- 0:20
data.frame(x, p1 = pnorm(x, lower.tail = FALSE), p2 = 1 - pnorm(x))
```

```
##     x           p1           p2
## 1   0 5.000000e-01 5.000000e-01
## 2   1 1.586553e-01 1.586553e-01
## 3   2 2.275013e-02 2.275013e-02
## 4   3 1.349898e-03 1.349898e-03
## 5   4 3.167124e-05 3.167124e-05
## 6   5 2.866516e-07 2.866516e-07
## 7   6 9.865876e-10 9.865877e-10
## 8   7 1.279813e-12 1.279865e-12
```

```
## 9   8 6.220961e-16 6.661338e-16
## 10  9 1.128588e-19 0.000000e+00
## 11 10 7.619853e-24 0.000000e+00
## 12 11 1.910660e-28 0.000000e+00
## 13 12 1.776482e-33 0.000000e+00
## 14 13 6.117164e-39 0.000000e+00
## 15 14 7.793537e-45 0.000000e+00
## 16 15 3.670966e-51 0.000000e+00
## 17 16 6.388754e-58 0.000000e+00
## 18 17 4.105996e-65 0.000000e+00
## 19 18 9.740949e-73 0.000000e+00
## 20 19 8.527224e-81 0.000000e+00
## 21 20 2.753624e-89 0.000000e+00
```

Of course, we can use the symmetry of the normal distribution to compute these without catastrophic cancellation and without `lower.tail = FALSE`

```
x <- 7:12
data.frame(x, p1 = pnorm(x, lower.tail = FALSE), p2 = pnorm(- x))
```

```
##    x           p1           p2
## 1  7 1.279813e-12 1.279813e-12
## 2  8 6.220961e-16 6.220961e-16
## 3  9 1.128588e-19 1.128588e-19
## 4 10 7.619853e-24 7.619853e-24
## 5 11 1.910660e-28 1.910660e-28
## 6 12 1.776482e-33 1.776482e-33
```

but for nonsymmetric distributions, `lower.tail = FALSE` is essential for avoiding catastrophic cancellation for upper tail probabilities.

The same argument works the same way for quantiles.

```
p <- 10^(-(1:20))
cbind(p = p, q1 = qnorm(p, lower.tail = FALSE), q2 = qnorm(1 - p))
```

```
##           p       q1       q2
##  [1,] 1e-01 1.281552 1.281552
##  [2,] 1e-02 2.326348 2.326348
##  [3,] 1e-03 3.090232 3.090232
##  [4,] 1e-04 3.719016 3.719016
##  [5,] 1e-05 4.264891 4.264891
##  [6,] 1e-06 4.753424 4.753424
##  [7,] 1e-07 5.199338 5.199338
##  [8,] 1e-08 5.612001 5.612001
##  [9,] 1e-09 5.997807 5.997807
## [10,] 1e-10 6.361341 6.361341
## [11,] 1e-11 6.706023 6.706023
## [12,] 1e-12 7.034484 7.034487
## [13,] 1e-13 7.348796 7.348755
## [14,] 1e-14 7.650628 7.650731
## [15,] 1e-15 7.941345 7.941444
## [16,] 1e-16 8.222082 8.209536
## [17,] 1e-17 8.493793      Inf
## [18,] 1e-18 8.757290      Inf
## [19,] 1e-19 9.013271      Inf
## [20,] 1e-20 9.262340      Inf
```

7.2 The Machine Epsilon

With *real* real numbers for every $\varepsilon > 0$ we have $1 + \varepsilon > 1$. Not so with computer arithmetic.

```
foo <- 1 + 1e-100
identical(foo, 1)
```

```
## [1] TRUE
```

According to `?.Machine`

```
.Machine$double.eps
```

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

is "the smallest positive floating-point number `x` such that `1 + x != 1`". According to the Wikipedia page for "machine epsilon" definitions of this concept vary among different authorities, but the one R uses is widely used and is also the definition used by C and C++.

The C program

```
#include <float.h>
#include <stdio.h>

int main(void)
{
    printf("machine epsilon: %e\n", DBL_EPSILON);
    return 0;
}
```

and the C++ program

```
#include <limits>
#include <iostream>
using namespace std;

int main()
{
    cout << "machine epsilon:" <<
        std::numeric_limits<double>::epsilon() << endl;
    return 0;
}
```

print the same number as R does above.

Is the definition correct?

```
epsilon <- .Machine$double.eps
1 + epsilon == 1
```

```
## [1] FALSE
```

```
1 + 0.9 * epsilon == 1
```

```
## [1] FALSE
```

```
1 + 0.8 * epsilon == 1
```

```
## [1] FALSE
```

```
1 + 0.7 * epsilon == 1
```

```
## [1] FALSE
```

```
1 + 0.6 * epsilon == 1
```

```
## [1] FALSE
```

```
1 + 0.5 * epsilon == 1
```

```
## [1] TRUE
```

```
1 + 0.4 * epsilon == 1
```

```
## [1] TRUE
```

Hmmmmmmmmmm. It appears that the "definition" in the R documentation is actually wrong. Perhaps, they are using one of the other definitions that Wikipeda mentions. Oh. The C11 standard says `DBL_EPSILON` is "the difference between 1 and the least value greater than 1 that is representable in the given floating point type, b^{1-p}." I guess that that means that `DBL_EPSILON` (hence the rest too) has to be a power of 2.

```
log2(epsilon)
```

```
## [1] -52
```

So that seems right.

Anyway, all of these technicalities aside, the machine epsilon is *more or less* the relative precision of computer arithmetic.

R uses it to define things like tolerances

```
args(all.equal.numeric)
```

```
## function (target, current, tolerance = sqrt(.Machine$double.eps),
##     scale = NULL, countEQ = FALSE, formatFUN = function(err,
##         what) format(err), ..., check.attributes = TRUE)
## NULL
```

And you should too. You should also follow this example in making tolerance(s) an argument of your functions (that need tolerances) so the user can override your default.

Also, returning to the preceding section, we see that the machine epsilon is where the complement rule starts to fail.

7.3 The Short-Cut Formula for Variance Fails

What some intro stats books call the "short-cut" formula for variance

$$\operatorname{var}(X) = E(X^2) - E(X)^2$$

is a mathematical identity when using *real* real numbers. It is an invitation to catastrophic cancellation when using computer arithmetic.

Always use the two-pass algorithm

$$\bar{x}_n = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$v_n = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x}_n)^2$$

```
x <- 1:10
# short cut
mean(x^2) - mean(x)^2
```

```
## [1] 8.25
```

```
# two pass
moo <- mean(x)
mean((x - moo)^2)
```

```
## [1] 8.25
```

Looks OK. What's the problem? But

```
x <- x + 1e9
# short cut
mean(x^2) - mean(x)^2
```

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

```
# two pass
moo <- mean(x)
mean((x - moo)^2)
```

```
## [1] 8.25
```

Catastrophic cancellation!

There is also sophisticated one-pass algorithm (Chan, Golub, and LeVeque (1983), "Algorithms for computing the sample variance: Analysis and recommendations", *American Statistician*, 37, 242-247), but it is not efficient in R (it can be used when you are programming in C or C++).

7.4 Special Functions

7.4.1 Log and Exp

Some commonly used mathematical operations invite catastrophic cancellation. R and C and C++ provide special functions to do these right.

The R function `log1p` calculates $\log(1 + x)$ in a way that avoids catastrophic cancellation when x is nearly zero. We know from calculus (Taylor series) that $\log(1 + x) \approx x$ for small x.

```
log1p(1 / pi)
```

```
## [1] 0.2763505
```

```
log(1 + 1 / pi)
```

```
## [1] 0.2763505
```

not much difference, but

```
foo <- 1e-20
log1p(foo)
```

```
## [1] 1e-20
```

```
log(1 + foo)
```

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

catastrophic cancellation!

The R function `expm1` calculates $e^x - 1$ in a way that avoids catastrophic cancellation when x is nearly zero. We know from calculus (Taylor series) that $e^x - 1 \approx x$ for small x.

```
expm1(1 / pi)
```

```
## [1] 0.3748022
```

```
exp(1 / pi) - 1
```

```
## [1] 0.3748022
```

not much difference, but

```
foo <- 1e-20
expm1(foo)
```

```
## [1] 1e-20
```

```
exp(foo) - 1
```

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

catastrophic cancellation!

C and C++ also have `log1p` and `expm1`. In fact, R is just calling the C functions to do them.

7.4.2 Trig Functions

New in R-3.1.0 are functions `cospi(x)`, `sinpi(x)`, and `tanpi(x)`, which compute `cos(pi*x)`, `sin(pi*x)`, and `tan(pi*x)`.
These functions are also in C and C++.

```
x <- (0:20) / 2
data.frame(sin = sin(pi * x), sinpi = sinpi(x))
```

```
##              sin sinpi
## 1   0.000000e+00     0
## 2   1.000000e+00     1
## 3   1.224647e-16     0
## 4  -1.000000e+00    -1
## 5  -2.449294e-16     0
## 6   1.000000e+00     1
## 7   3.673940e-16     0
## 8  -1.000000e+00    -1
## 9  -4.898587e-16     0
## 10  1.000000e+00     1
## 11  6.123234e-16     0
## 12 -1.000000e+00    -1
## 13 -7.347881e-16     0
## 14  1.000000e+00     1
## 15  8.572528e-16     0
## 16 -1.000000e+00    -1
## 17 -9.797174e-16     0
## 18  1.000000e+00     1
## 19  1.102182e-15     0
## 20 -1.000000e+00    -1
## 21 -1.224647e-15     0
```

8 A Problem Requiring Care

8.1 Introduction

The log likelihood for the usual parameter p for the binomial distribution with observed data x and sample size n is

$$l(p) = x \log(p) + n \log(1 - p)$$

In terms of the "natural" parameter

$$\theta = \operatorname{logit}(p) = \log(p) - \log(1 - p)$$

the log likelihood is

$$l(\theta) = x\theta - n \log(1 + e^\theta)$$

The function going the other way between p and θ is

$$p = \frac{e^\theta}{1 + e^\theta} = \frac{1}{e^{-\theta} + 1}$$

The first derivative of the log likelihood is

$$l'(\theta) = x - n \frac{e^\theta}{1 + e^\theta} = x - np$$

and the second derivative is

$$l''(\theta) = -n \frac{e^\theta}{1 + e^\theta} + n \frac{(e^\theta)^2}{(1 + e^\theta)^2} = -np(1 - p)$$

we want an R function that evaluates this log likelihood and its derivatives.

8.2 Design

The first problem we have to deal with is overflow. We never want e^θ or $e^{-\theta}$ to overflow. We see that we can write p in terms of either e^θ or $e^{-\theta}$, so we want to pick the expression that cannot overflow. Similarly the expression for the log likelihood itself can be rewritten in terms of $e^{-\theta}$

$$l(\theta) = x\theta - n \log[e^\theta (e^{-\theta} + 1)] = x\theta - n\theta - n \log(e^{-\theta} + 1) = -(n - x)\theta - n \log(e^{-\theta} + 1)$$

and here too we want to pick the expression that cannot overflow.

The second problem we want to deal with is catastrophic cancellation. We never want to evaluate $1 - p$ by subtracting p from 1. Instead use algebra to rewrite it so there is no subtraction

$$q = 1 - p = 1 - \frac{e^\theta}{1 + e^\theta} = \frac{1}{1 + e^\theta} = \frac{e^{-\theta}}{e^{-\theta} + 1}$$

so now there is no catastrophic cancellation here and no overflow either if we choose the expression that does not overflow.

8.3 Implementation

```
logl <- function(theta, x, n, deriv = 2) {
    stopifnot(is.numeric(theta))
    stopifnot(is.finite(theta))
    stopifnot(length(theta) == 1)
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 1)
    if (x != round(x)) stop("x must be integer")
    stopifnot(is.numeric(n))
    stopifnot(is.finite(n))
    stopifnot(length(n) == 1)
    if (n != round(n)) stop("n must be integer")
    stopifnot(x <= n)
    stopifnot(length(deriv) == 1)
    stopifnot(deriv %in% 0:2)
    val <- if (theta < 0) x * theta - n * log1p(exp(theta)) else
        - (n - x) * theta - n * log1p(exp(- theta))
}
```

Note that we use `log1p` in the obvious places to avoid catastrophic cancellation.

8.4 Test

For once we won't test that every error message works as supposed. We leave that as an exercise for the reader.

Our function doesn't do derivatives yet, but we want to get to testing right away.

```
thetas <- seq(-10, 10)
x <- 0
n <- 10
log.thetas <- Map(function(theta) logl(theta, x, n), thetas)
log.thetas.too <- Map(function(theta) dbinom(x, n,
    1 / (exp(- theta) + 1), log = TRUE), thetas)
all.equal(log.thetas, log.thetas.too)
```

```
## [1] TRUE
```

8.5 More Design

The first derivative is simple, but we worry about catastrophic cancellation in $x - np$. We special-case one case: when $x = n$ we have

$$l'(\theta) = n(1 - p) = nq$$

and we want to be sure to evaluate q without catastrophic cancellation.

But for the general case, there does not seem to be any way to avoid cancellation (maybe we shouldn't call it "catastrophic" here) if it occurs. We have to compare x and np somehow, and comparing "real" (`double`) numbers is always fraught with danger (or at least inaccuracy).

8.6 Re-Implementation

```
logl <- function(theta, x, n, deriv = 2) {
    stopifnot(is.numeric(theta))
    stopifnot(is.finite(theta))
    stopifnot(length(theta) == 1)
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 1)
    if (x != round(x)) stop("x must be integer")
    stopifnot(is.numeric(n))
    stopifnot(is.finite(n))
    stopifnot(length(n) == 1)
    if (n != round(n)) stop("n must be integer")
    stopifnot(x <= n)
    stopifnot(length(deriv) == 1)
    stopifnot(deriv %in% 0:2)
    val <- if (theta < 0) x * theta - n * log1p(exp(theta)) else
        - (n - x) * theta - n * log1p(exp(- theta))
    result <- list(value = val)
    if (deriv == 0) return
    pp <- if (theta < 0) exp(theta) / (1 + exp(theta)) else
        1 / (exp(- theta) + 1)
    qq <- if (theta < 0) 1 / (1 + exp(theta)) else
        exp(- theta) / (exp(- theta) + 1)
    grad <- if (x < n) x - n * pp else n * qq
    result$gradient <- grad
    result
}
```

8.7 More Tests

Now that we know the first part of our function (log likelihood calculation) is correct, we can trust it while we are testing whether the derivative is correct.

8.7.1 Derivatives Computed by R

I can think of two obvious methods of testing derivatives. One is to use R's knowledge of calculus, which is primitive but good enough for this problem.

```
d1 <- D(expression(x * theta - n * log(1 + exp(theta))), "theta")
d1
```

```
## x - n * (exp(theta)/(1 + exp(theta)))
```

```
mygrad <- function(theta, x, n) eval(d1)
g0 <- Map(function(theta) logl(theta, x, n)$gradient, thetas)
g1 <- Map(function(theta) mygrad(theta, x, n), thetas)
all.equal(g0, g1)
```

```
## [1] TRUE
```

8.7.2 Derivatives Computed by Numerical Differentiation

Even when R does not know how to check derivatives, they can still be approximated numerically. The simplest way is to use finite differences

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}, \qquad \text{for small } h$$

but there is a CRAN package `numDeriv` that does a lot more sophisticated calculations.

```
library(numDeriv)
numgrad <- function(theta) grad(function(theta) logl(theta, x, n)$value, theta)
g2 <- Map(numgrad, thetas)
all.equal(g0, g2)
```

```
## [1] TRUE
```

The definition of `numgrad` above may seem confusing: too many `thetas`! First, `numgrad` is itself a function of `theta`. It is supposed to calculate the first derivative of the log likelihood $l'(\theta)$. We calculate that using the R function `grad` in the R package `numDeriv`. It wants a function as its first argument (the function to differentiate). That function, too, we think of as a function of `theta`, and we define that function right there as an anonymous expression

```
function(theta) logl(theta, x, n)$value
```

and `theta` in this expression has nothing whatsoever to do with `theta` outside this expression (just like any argument of any function). In this expression `theta` is the argument of this anonymous function. It might help readability to rewrite our definition of `numgrad` as

```
function(theta) grad(function(theta.too) logl(theta.too, x, n)$value, theta)
```

so we can tell our `thetas` apart, but R has no trouble with the way it was written first.

The last `theta` in the definition of `numgrad` is the point where `grad` is to evaluate the derivative.

8.7.3 More Derivatives Computed by Numerical Differentiation

We also need to test the special case `x == n` and while we are at it, it wouldn't hurt to test the special case `x == 0`.

```
g0 <- Map(function(theta) logl(theta, n, n)$gradient, thetas)
g1 <- Map(function(theta) mygrad(theta, n, n), thetas)
all.equal(g0, g1)
```

```
## [1] TRUE
```

```
numgrad <- function(theta) grad(function(theta) logl(theta, n, n)$value, theta)
g2 <- Map(numgrad, thetas)
all.equal(g0, g2)
```

```
## [1] TRUE
```

```
g0 <- Map(function(theta) logl(theta, 0, n)$gradient, thetas)
g1 <- Map(function(theta) mygrad(theta, 0, n), thetas)
all.equal(g0, g1)
```

```
## [1] TRUE
```

```
numgrad <- function(theta) grad(function(theta) logl(theta, 0, n)$value, theta)
g2 <- Map(numgrad, thetas)
all.equal(g0, g2)
```

```
## [1] TRUE
```

It is a bit ugly that our tests have to redefine `numgrad` each time, but it doesn't matter because no one has to use the tests, just the function `logl` that we are testing.

8.8 Still More Design

The second derivative is even simpler,

$$l''(\theta) = -npq$$

so long as we calculate p and q without catastrophic cancellation, which we know how to do.

8.9 Re-re-Implementation

```
logl <- function(theta, x, n, deriv = 2) {
    stopifnot(is.numeric(theta))
    stopifnot(is.finite(theta))
    stopifnot(length(theta) == 1)
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 1)
    if (x != round(x)) stop("x must be integer")
    stopifnot(is.numeric(n))
    stopifnot(is.finite(n))
    stopifnot(length(n) == 1)
    if (n != round(n)) stop("n must be integer")
    stopifnot(0 <= x)
    stopifnot(x <= n)
    stopifnot(length(deriv) == 1)
    stopifnot(deriv %in% 0:2)
    val <- if (theta < 0) x * theta - n * log1p(exp(theta)) else
        - (n - x) * theta - n * log1p(exp(- theta))
    result <- list(value = val)
    if (deriv == 0) return(result)
    pp <- if (theta < 0) exp(theta) / (1 + exp(theta)) else
        1 / (exp(- theta) + 1)
    qq <- if (theta < 0) 1 / (1 + exp(theta)) else
        exp(- theta) / (exp(- theta) + 1)
    grad <- if (x < n) x - n * pp else n * qq
    result$gradient <- grad
    if (deriv == 1) return(result)
    result$hessian <- (- n * pp * qq)
    return(result)
}
```

I noticed in this re-re-implementation that our re-implementation was completely broken in a way that was not tested. It did not return the right thing in case `deriv = 0`. Now this is fixed, but we should be sure to test it this time.

Much later (during class) I noticed that I was missing the test that `0 <= x` so that has been added also.

8.10 Still More Tests

```
logl(1.1, x, n, 0)
```

```
## $value
## [1] -13.87335
```

```
logl(1.1, x, n, 1)
```

```
## $value
## [1] -13.87335
##
## $gradient
## [1] -7.502601
```

```
logl(1.1, x, n, 2)
```

```
## $value
## [1] -13.87335
##
## $gradient
## [1] -7.502601
##
## $hessian
## [1] -1.873699
```

```
logl(1.1, x, n, 3)
```

```
## Error in logl(1.1, x, n, 3): deriv %in% 0:2 is not TRUE
```

So we see the `deriv` argument (now) works correctly.

We still have to test the second derivative. We do this just like we tested the first derivative.

```
d2 <- D(d1, "theta")
d2
```

```
## -(n * (exp(theta)/(1 + exp(theta)) - exp(theta) * exp(theta)/(1 +
##     exp(theta))^2))
```

```
myhess <- function(theta, x, n) eval(d2)
h0 <- Map(function(theta) logl(theta, x, n)$hessian, thetas)
h1 <- Map(function(theta) myhess(theta, x, n), thetas)
all.equal(h0, h1)
```

```
## [1] TRUE
```

```
numhess <- function(theta)
    grad(function(theta) logl(theta, x, n)$gradient, theta)
h2 <- Map(numhess, thetas)
all.equal(h0, h2)
```

```
## [1] TRUE
```

Everything looks good.

8.11 One Final Comment

We could replace the test function

```
function(theta, x) dbinom(x, 20, prob = 1 / (1 + exp(- theta)), log = TRUE)
```

that appears in problem 7 on homework 1 with our new improved version

```
function(theta, x) logl(theta, x, 20, deriv = 0)$value
```

(but I haven't actually tested that, so I'm not 100% certain of that).

9 Another Problem Requiring Care

9.1 Introduction

Suppose we have a probability density function (PDF) or probability mass function (PMF) of the form

$$f_\theta(x) = a(\theta) b(x) e^{x\theta}$$

(in which case this is called an *exponential family of distributions*), and

1. we do not know how to calculate the function a but
2. we do know how to simulate random variables having this distribution.

This may seem crazy, but there is a general methodology for simulating probability distributions known only up to an unknown normalizing constant called the Metropolis algorithm (Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953), "Equation of state calculations by fast computing machines", *Journal of Chemical Physics*, 21, 1087-1092).

Geyer and Thompson (1992, "Constrained Monte Carlo maximum likelihood for dependent data (with discussion), *Journal of the Royal Statistical Society, Series B*, 54, 657-699) show that the following method approximates the log likelihood of this distribution, when x_{obs} is the observed data, x is a vector of simulations of the distribution for parameter value ψ,

$$l(\theta) = x_{\text{obs}}\theta - \log\left(\sum_{i=1}^{n} e^{x_i(\theta - \psi)}\right)$$

or in R

```
logl <- function(theta) xobs * theta - log(sum(exp(x * (theta - psi))))
```

except that won't work well because the exponentials are likely to overflow or underflow.

9.2 Design

Our problem is to rewrite this so none of exponentials overflow and at least some of the exponentials do not underflow.

The key idea is to add and subtract a constant from each exponential.

$$\begin{aligned}
\log\left(\sum_{i=1}^n e^{x_i(\theta-\psi)}\right) &= \log\left(\sum_{i=1}^n e^{x_i(\theta-\psi)+c-c}\right) \\
&= \log\left(\sum_{i=1}^n e^c e^{x_i(\theta-\psi)-c}\right) \\
&= \log\left(e^c \sum_{i=1}^n e^{x_i(\theta-\psi)-c}\right) \\
&= \log(e^c) + \log\left(\sum_{i=1}^n e^{x_i(\theta-\psi)-c}\right) \\
&= c + \log\left(\sum_{i=1}^n e^{x_i(\theta-\psi)-c}\right)
\end{aligned}$$

This is true for any real number c, but we need to choose c so we know the exponentials cannot overflow. An obvious choice is to choose c to be the largest of the terms $x_i(\theta - \psi)$.

This will make the largest term in the sum equal to one, so not all of the exponentials underflow (and those that do make negligible contribution to the sum).

Having decided to make one term in the sum equal to one, we now have an opportunity to use `log1p` to calculate the log. And we should to avoid catastrophic cancellation.

9.3 Implementation

Before we start this problem, we clean up the R global environment

```
rm(list = ls())
```

We write the log likelihood as

```
logl <- function(theta) {
     foo <- x * (theta - psi)
     foomax <- max(foo)
     i <- which(foo == foomax)
     i <- i[1] # just in case there was more than one largest term
     foo <- foo[-i]
     bar <- foomax + log1p(sum(exp(foo - foomax)))
     xobs * theta - bar
}
```

For once we dispense with GIEMO and write the function using global variables as explained in Section 7.4.2 of the "Basics" handout.

9.4 Tests

We happen to have some appropriate data for this problem.

```
load(url("http://www.stat.umn.edu/geyer/3701/data/ising.rda"))
ls()
```

```
## [1] "logl" "psi"  "x"    "xobs"
```

Note that it is necessary to use the `url` function here, whereas it is unnecessary when reading from a URL with `scan`, `read.table`, or `read.csv`, because the "read" functions do extra trickery to recognize URLS and do the right thing, and `load` doesn't bother.

What the model actually is, we won't bother to explain. It is irrelevant to the present discussion (avoiding overflow and catastrophic cancellation).

It turns out that this function, which was tricky enough to write, is even trickier to test because any other method I can think of to calculate this does not work because of either overflow or catastrophic cancellation.

So we just plot the function and see that it makes sense.

```
thetas <- seq(psi / 1.005, psi * 1.005, length = 101)
l0 <- Map(logl, thetas)
plot(thetas, unlist(l0), xlab=expression(theta), ylab=expression(l(theta)))
```

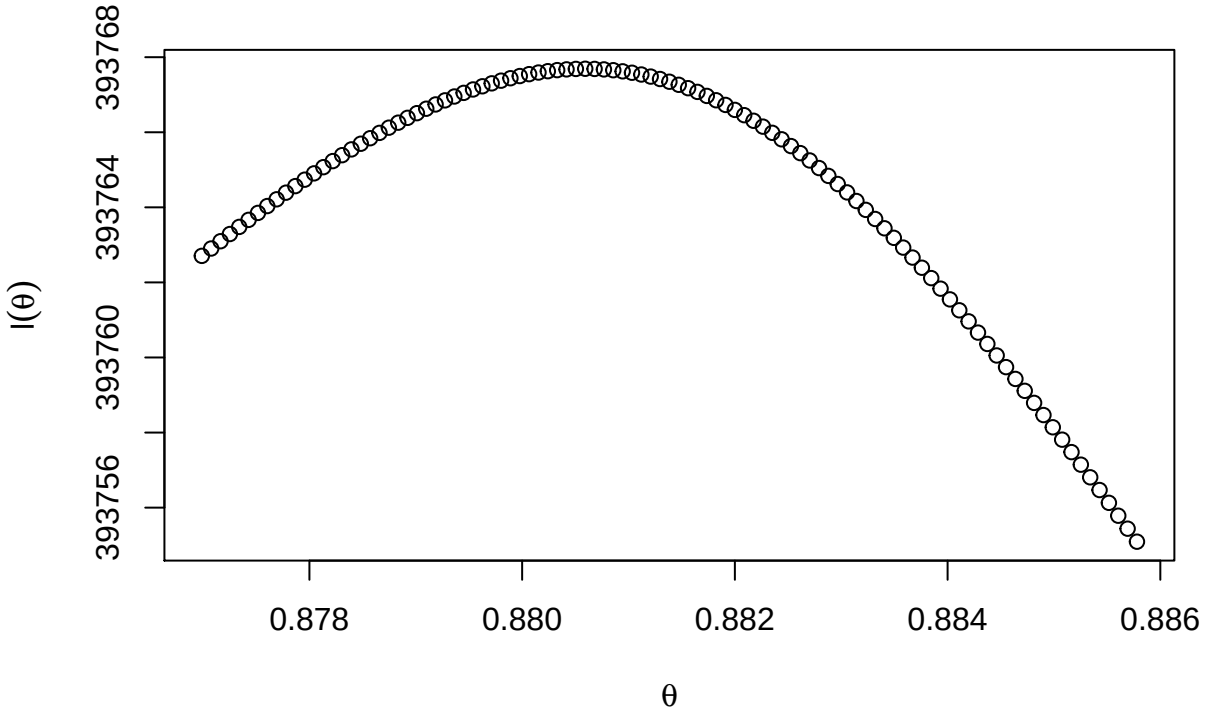

Figure 1: Log Likelihood Function

Theory says that this function should be concave and asymptotically linear, that is, bends downward and looks like a linear function for very large (positive or negative) values of the argument. At least it looks like that.