Stat 3701 Lecture Notes: Optimization and Solving Equations

Charles J. Geyer

November 27, 2022

1 License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License (http://creativecommons.org/licenses/by-sa/4.0/).

2 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.17.
- The version of the `trust` package used to make this document is 0.1.8.
- The version of the `alabama` package used to make this document is 2022.4.1.
- The version of the `numDeriv` package used to make this document is 2016.8.1.1.
- The version of the `MASS` package used to make this document is 7.3.58.1.

3 Optimization

Optimization is a very big subject, and R has a lot of different functions in a lot of different packages that do optimization. So we have to look at just a few issues.

4 Solving Equations

We know from calculus that to find an optimum (maximum or minimum) of a smooth function on a region without boundaries, look for a point where the derivative is zero.

So having a method of solving equations can be a method of doing optimization. But no good method of optimization just finds points where the first derivative is zero. That can be *part* of the method, but any good method also assures that the function goes downhill if minimizing and uphill if maximizing.

But sometimes the problem is to solve an equation, not to optimize some function. Then you need an equation solver, also called a root finder.

So we really have two kinds of problems, optimization and root finding, where calculus sees only one. In general, optimization is much easier than root finding, and most optimization software only does optimization, not root finding.

So kinds of optimization problems, particularly nonsmooth optimization (optimization of non-differentiable functions) and inequality constrained optimization have no simple analogs in root finding.

5 CRAN Task Views

CRAN has a task view on Optimization and Mathematical Programming.

The “programming” here does not mean writing algorithms in computer language (what this course is about). Instead it refers to inequality constrained optimization, as in the terms “linear programming (optimizing a linear function subject to linear equality and inequality constraints) and”quadratic programming" (optimizing a quadratic function subject to linear equality and inequality constraints). This “programming” terminology is a bit dated, but one does see it.

This task view covers *a lot* of packages and functions.

CRAN also has a task view on Numerical Mathematics that has a (short) section on general root finding (only two CRAN packages) and a somewhat longer section on finding roots of polynomials.

6 One-Dimensional Optimization and Root Finding

6.1 Optimization

The special case of one-dimensional optimization is much easier than two-and-higher-dimensional because one can just graph the objective function and do (crude) optimization by eye.

It is also easier for the computer because it can do “bracketing”. We know from calculus that, if f is a continuous function and $f(r) < 0$ and $f(l) > 0$, then there is a point x between r and l such that $f(x) = 0$.

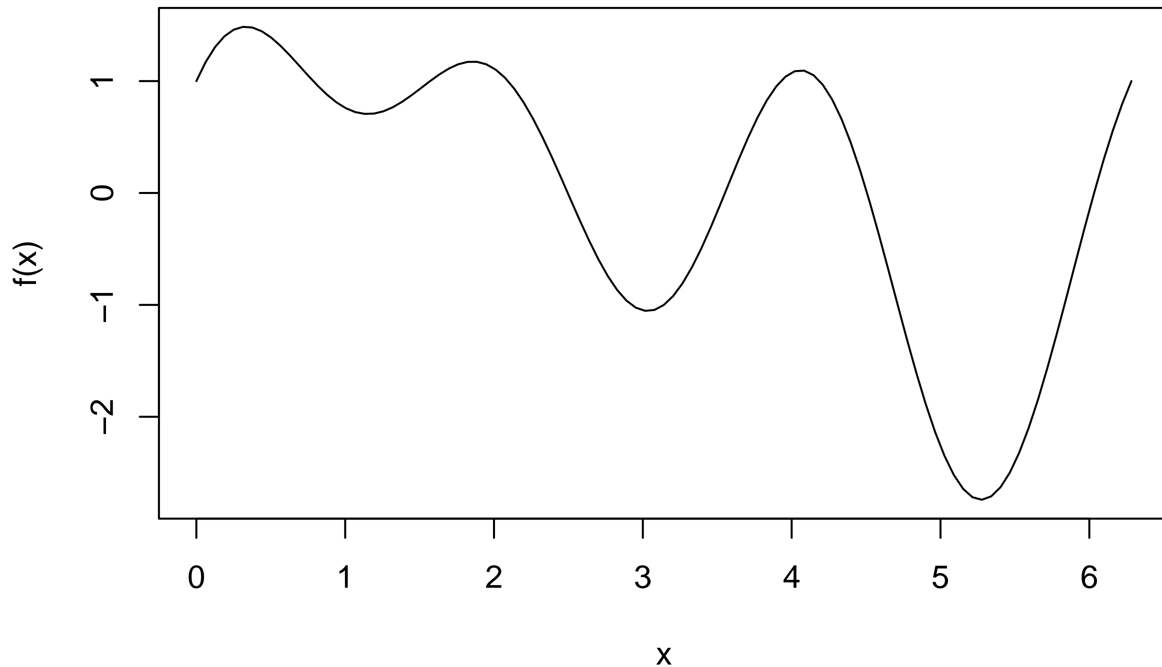
So as long as one has a solution bracketed, it is easy for the computer to keep subdividing the interval so as to keep a solution bracketed.

If we continue the above assumptions and evaluate $f(y)$ and find $f(y) > 0$, then we know that there is a zero in the interval (r, y). Conversely, if we find $f(y) < 0$, then there is a zero in the interval (y, l).

There are two R functions in the R core for one-dimensional optimization (`optimize`) and one-dimensional root finding (`uniroot`). The former we have already met in course notes on basics of R (Sections 4.4, 4.5, and 7.4). The latter works similarly.

Let’s try both out on a simple example.

```
f <- function(x) sin(x) + sin(2 * x) + cos(3 * x)
curve(f, from = 0, to = 2 * pi)
```



First optimize

```
optimize(f, interval = c(0, 2 * pi))
```

```
## $minimum
## [1] 3.033129
##
## $objective
## [1] -1.054505
```

We see it finds a *local minimum*, a point where the first derivative is zero and the second derivative is positive, a point where the value is lower and at any other point in some neighborhood of this point, but it does not find the *global minimum*, the point where the value is lowest (in this interval — of course, this is a periodic function with period 2π, so if x is a local or global minimum, then $x + 2\pi k$ is also a local or global minimum, respectively, for any integer k).

The global minimum, we can see from the picture is between 4 and 2π

```
optimize(f, interval = c(4, 2 * pi))
```

```
## $minimum
## [1] 5.273383
##
## $objective
## [1] -2.741405
```

6.2 Root Finding

Now for root finding

```
uniroot(f, interval = c(0, 2 * pi))
```

```
## Error in uniroot(f, interval = c(0, 2 * pi)): f() values at end points not of opposite sign
```

Right. Its complaint is correct. We can see from the picture that the function is positive at zero and negative at 6.

```
uniroot(f, interval = c(0, 6))
```

```
## $root
## [1] 2.495467
##
## $f.root
## [1] 7.797267e-07
##
## $iter
## [1] 7
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

So it works, but we had to use the knowledge obtained by eyeballing the graph to make it work.

Also notice that it finds *a* root not *all* the roots. There are other roots (as again can be seen in the picture).

```
uniroot(f, interval = c(3, 4))$root
```

```
## [1] 3.552112
```

```
uniroot(f, interval = c(4, 5))$root
```

```
## [1] 4.506593
```

```
uniroot(f, interval = c(5, 2 * pi))$root
```

```
## [1] 6.032163
```

Although `uniroot` needs an interval specified, an optional argument allows it to keep going if the interval is incorrect.

```
uout <- uniroot(f, interval = c(0, 2 * pi), extendInt = "yes")
uout$root
```

```
## [1] 21.34502
```

Since we know f is periodic with period 2π, where is the corresponding root between 0 and 2π?

```
foo <- uout$root / (2 * pi)
foo <- foo %% 1
foo <- 2 * pi * foo
foo
```

```
## [1] 2.495463
```

```
f(foo)
```

```
## [1] 1.246467e-05
```

Another optional argument allows us to specify more accuracy.

```
uniroot(f, interval = c(0, 6), tol = 1e-10)
```

```
## $root
## [1] 2.495467
##
## $f.root
## [1] -4.075074e-13
##
## $iter
## [1] 8
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 5.000089e-11
```

6.3 Newton's Method

`optimize` also has a `tol` argument but no optional argument analogous to the `extendInt` argument of `uniroot`.

```
x1 <- optimize(f, interval = c(4, 2 * pi))$minimum
x2 <- optimize(f, interval = c(4, 2 * pi), tol = 1e-10)$minimum
x1
```

```
## [1] 5.273383
```

```
x2
```

```
## [1] 5.273376
```

So how much improvement did we get? For that we have to look at the derivative.

```
g <- D(expression(sin(x) + sin(2 * x) + cos(3 * x)), "x")
g
```

```
## cos(x) + cos(2 * x) * 2 - sin(3 * x) * 3
```

```
gfun <- function(x) eval(g)
gfun(x1)
```

```
## [1] 9.439912e-05
```

```
gfun(x2)
```

```
## [1] 1.328198e-07
```

It is clear that `x2` is closer to the minimum than `x1` (the derivative is closer to zero), but it is not clear exactly what the `tol` argument specifies. From the description "desired accuracy" in the help, it is perhaps the distance from the actual minimum.

Applying Newton's method to our putative solution, gives perhaps a more accurate estimate. Newton's method (also called Newton-Raphson) is a method of root finding, which to do optimization we apply to the first derivative. To find a point where $g(x) = 0$, expand g around a point x_0 in a Taylor series

$$g(x) \approx g(x_0) + g'(x_0)(x - x_0)$$

and set equal to zero and solve

$$
\begin{aligned}
0 &= g(x_0) + g'(x_0)(x - x_0) \\
-g(x_0) &= g'(x_0)(x - x_0) \\
-\frac{g(x_0)}{g'(x_0)} &= x - x_0 \\
x &= x_0 - \frac{g(x_0)}{g'(x_0)}
\end{aligned}
$$

so

```
h <- D(g, "x")
hfun <- function(x) eval(h)
x3 <- x2 - gfun(x2) / hfun(x2)
gfun(x2)
```

```
## [1] 1.328198e-07
```

```
gfun(x3)
```

```
## [1] 1.554312e-15
```

```
x2 - x3
```

```
## [1] 9.916358e-09
```

So the result of `optimize` with `tol = 1e-10` is about 9.92×10^{-9} away from the actual minimum (as close as we can come with the accuracy of computer arithmetic). So it is still not clear what `tol` is doing.

In summary, one-dimensional optimization and root finding is "easy" because graphing the function can guide solution. But there is still a lot to say about it.

Everything is much harder in higher dimensions because the graph of the function is so much harder (perhaps even impossible) to visualize.

7 Two-Dimensional Optimization and Root Finding

7.1 Optimization

7.1.1 R function `nlm`

At least in two-dimensions we can still graph the function, although much more messily.

```
f <- function(x) {
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 2)
    x1 <- x[1]
    x2 <- x[2]
    sin(x1) + sin(x2) + cos(3 * x1 * x2) + x1^2 + x2^2
}
```

Since we know that sines and cosines are bounded in absolute value by one, we can see that $f(x) \geq 1$ when $|x_i| \geq 2$, for $i = 1, 2$. So if there is a minimum below 1, it must occur in this box.

Try to plot the function using the R function `contour`. Since `contour` does not take

```
x <- seq(-2, 2, length = 101)
xx <- rep(x, each = 101)
head(xx)
```

```
## [1] -2 -2 -2 -2 -2 -2
```

```r
yy <- rep(x, times = 101)
head(yy)
```

```
## [1] -2.00 -1.96 -1.92 -1.88 -1.84 -1.80
```

```r
sqrt(length(xx))
```

```
## [1] 101
```

```r
sqrt(length(yy))
```

```
## [1] 101
```

```r
zz <- apply(cbind(xx, yy), 1, f)
class(zz)
```

```
## [1] "numeric"
```

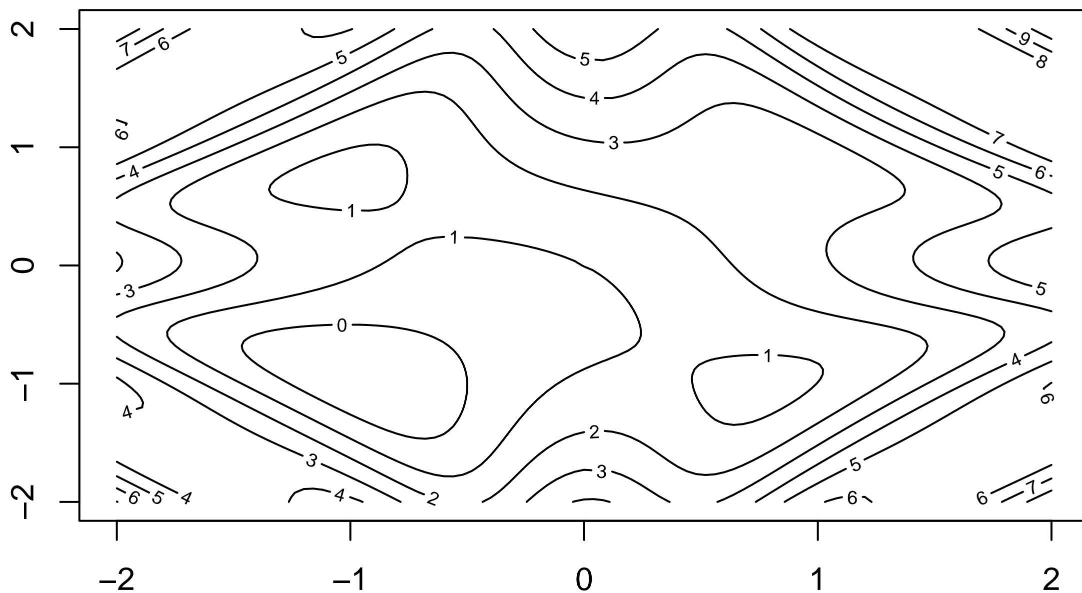
```r
sqrt(length(yy))
```

```
## [1] 101
```

```r
range(zz)
```

```
## [1] -0.7281865 10.6624488
```

```r
zz <- matrix(zz, 101)
contour(x, x, zz)
```



We see that there are regions where the function is less than one, so the global minimum does occur in this box. We also see that there are multiple local minima.

The simplest R function that does global minimization is `nlm`. Ideally we should provide derivatives for the objective function if we can.

```
ftoo <- deriv(expression(sin(x1) + sin(x2) + cos(3 * x1 * x2) + x1^2 + x2^2),
    namevec = c("x1", "x2"), function.arg = TRUE)
args(ftoo)
```

```
## function (x1, x2)
## NULL
```

```
ftoo(0, 0)
```

```
## [1] 1
## attr(,"gradient")
##      x1 x2
## [1,]  1  1
```

That is annoying: `deriv` produces a function of two arguments but `nlm` wants a function with one argument (which is a vector of length 2).

```
ftootoo <- function(x) {
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 2)
    x1 <- x[1]
    x2 <- x[2]
    ftoo(x1, x2)
}
ftootoo(c(0, 0))
```

```
## [1] 1
## attr(,"gradient")
##      x1 x2
## [1,]  1  1
```

Now we are ready to go.

```
nout <- nlm(ftootoo, c(0, 0))
nout
```

```
## $minimum
## [1] -0.7299735
##
## $estimate
## [1] -0.9425063 -0.9425063
##
## $gradient
## [1] 1.407247e-09 1.407247e-09
##
## $code
## [1] 1
##
## $iterations
## [1] 5
```

As we can see from our analysis (the global minimum has to be in this box) and from computation (the

minimum over the fairly finely space grid filling this box is

```r
min(zz)
```

```
## [1] -0.7281865
```

which is only a little bigger than the minimum found by `nlm`.

But if we had given a different starting value, `nlm` could converge to a local minimum that is not the global minimum

```r
nout1 <- nlm(ftootoo, c(0.75, -1))
nout1
```

```
## $minimum
## [1] 0.767623
##
## $estimate
## [1]  0.7487547 -1.0472218
##
## $gradient
## [1] -5.483357e-08 -5.094336e-07
##
## $code
## [1] 1
##
## $iterations
## [1] 5
```

So we see that `nlm` is only a method of *local optimization* not *global optimization*.

If one cannot be bothered to calculate derivatives of the objective function, `nlm` can do them by finite differences. It just takes more iterations to converge, and the answer is not as accurate (so you should provide derivatives if you can, like above).

```r
nout2 <- nlm(f, c(0, 0))
nout2
```

```
## $minimum
## [1] -0.7299735
##
## $estimate
## [1] -0.9425066 -0.9425066
##
## $gradient
## [1] 1.554312e-09 1.443290e-09
##
## $code
## [1] 1
##
## $iterations
## [1] 5
```

```r
attr(ftootoo(nout$estimate), "gradient")
```

```
##                x1           x2
## [1,] 1.407247e-09 1.407247e-09
```

```r
nout$gradient
```

```
## [1] 1.407247e-09 1.407247e-09
```

```r
attr(ftootoo(nout2$estimate), "gradient")
```

```
##                 x1            x2
## [1,] -4.954636e-06 -4.955145e-06
```

```r
nout2$gradient
```

```
## [1] 1.554312e-09 1.443290e-09
```

Clearly smaller gradient at the solution when we use analytic derivatives when they are available. Note that `nlm` thinks the gradient is smaller than it actually is when it is forced to use finite differences instead of derivatives.

7.1.2 R function `optim`

Another minimization function in the R core is `optim`, which has a plethora of methods, none of which are obviously better than `nlm` at local optimization of smooth functions.

7.1.2.1 Method Nelder-Mead

In particular, its default method `"Nelder-Mead"` has almost nothing to recommend it. It is slow, produces only limited accuracy, and still does only local optimization. Method `"Nelder-Mead"` can be useful if the objective function is not smooth or cannot be calculated accurately, so finite differences will not work. I have used it for minimizing functions that do another minimization inside themselves (an consequently have only limited accuracy) in the R function `reaster` in the CRAN package `aster`, but even there it is only used to provide a starting value for more accurate optimization methods. So we won't even bother to use method `"Nelder-Mead"` on an example.

7.1.2.2 Method CG

The other methods of `optim` with one exception (next paragraph) only do local minimization of smooth functions and use finite differences if analytic derivatives are not provided. So they are competitors of `nlm` that do more or less the same thing but with different interface (and perhaps not so convenient interface, because it is not the one that `deriv` likes).

```r
optim(c(0, 0), f, function(x) attr(ftootoo(x), "gradient"),
    method = "CG")
```

```
## $par
## [1] -0.9425064 -0.9425064
##
## $value
## [1] -0.7299735
##
## $counts
## function gradient
##       59       15
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Of course, the call here would be simpler if one cannot be bothered to supply analytic derivatives.

```r
optim(c(0, 0), f, method = "CG")
```

```
## $par
## [1] -0.9425063 -0.9425063
##
## $value
## [1] -0.7299735
##
## $counts
## function gradient
##      120       21
##
## $convergence
## [1] 0
##
## $message
## NULL
```

7.1.2.3 Method SANN

One method of `optim` purports to do global optimization. I say purports because it can get out of local minima, it is not guaranteed to say at a local minimum if started there. This is method "SANN" which does *simulated annealing* an adaptive random search algorithm. It looks in random places but also changes the probability distribution for its random choices as it goes along and also has a preference for steps that go downhill rather than uphill, but can go uphill to get out of (shallow) local minima.

Let's try it with default values for all options.

```r
# start at previously found local minimum that is not global minimum
nout1$estimate
```

```
## [1]  0.7487547 -1.0472218
```

```r
optim(nout1$estimate, f, method = "SANN")
```

```
## $par
## [1] -0.9436176 -0.9361088
##
## $value
## [1] -0.7298053
##
## $counts
## function gradient
##    10000       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

It has no trouble finding (what we know is for this problem) the global minimum, even though started at a local minimum that is not the global minimum, where `nlm` and all other methods of `optim` would stop right at the start claiming the starting point was the solution.

But it made a lot of function evaluations and would need to make a lot more if the function was high-dimensional. Also one would need to fuss a lot with the options.

Simulated annealing comes with a theorem that says it always converges to the global minimum provided a lot of conditions are met, that are usually unverifiable or so hard to verify that no user would try. Also this convergence theory requires values of the options that would make the function take practically forever to converge (it would take longer than you are willing to wait). So no one ever operates simulated annealing as the theory requires. Thus it is, in practice, *not* guaranteed to converge to the global minimum.

7.1.2.4 All Methods

This last point applies also to `nlm` and all of the other methods of `optim` as well. The options are there to be used. All of them are necessary to get good performance on some optimization problems. We just happened to pick a particularly easy optimization problem for our example. For easy problems the default settings work. For hard problems you need to use the options (trivially so, because that's how we define "hard").

7.1.3 R function `trust`

Tooting my own horn, we briefly discuss the R function `trust` in the CRAN package `trust`. This function also only does local minimization, requires analytic first and second derivatives, and does not have defaults for all its arguments other than the function and starting position. But it does come with a guarantee that it always converges to a local minimum, even in the hardest problems. It was developed to do problems where starting far from the global minimum crashes most optimizers.

```
f3 <- deriv3(expression(sin(x1) + sin(x2) + cos(3 * x1 * x2) + x1^2 + x2^2),
    namevec = c("x1", "x2"), function.arg = TRUE)
f3(0, 0)
```

```
## [1] 1
## attr(,"gradient")
##      x1 x2
## [1,]  1  1
## attr(,"hessian")
## , , x1
##
##      x1 x2
## [1,]  2  0
##
## , , x2
##
##      x1 x2
## [1,]  0  2
```

```
f4 <- function(x) {
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 2)
    x1 <- x[1]
    x2 <- x[2]
    fout <- f3(x1, x2)
    list(value = as.numeric(fout), gradient = drop(attr(fout, "gradient")),
        hessian = drop(attr(fout, "hessian")))
}
library(trust)
trust(f4, c(0, 0), rinit = 1, rmax = 1)
```

```
## $value
## [1] -0.7299735
##
```

```
## $gradient
##            x1            x2
## -5.470291e-12 -5.470291e-12
##
## $hessian
##          x1       x2
## x1 9.912809 5.727391
## x2 5.727391 9.912809
##
## $argument
## [1] -0.9425063 -0.9425063
##
## $converged
## [1] TRUE
##
## $iterations
## [1] 8
```

This function is used as the default optimizer for the R function `aster` in the CRAN package `aster` and although `nlm` and `optim` can also be used, they do not work as well.

7.2 Root Finding

There is a CRAN package `nleqslv` that does multi-dimensional root finding. I have never had occasion to need this, so never even knew of its existence before writing these notes.

I have never had occasion to use such a package and cannot think of a statistical problem that needs it, but it is there in case such a problem arises.

8 Inequality-Constrained Optimization

8.1 A Problem

The problem we will try to solve is calculating a likelihood-based confidence interval for the toy data we used for GLM examples in the course notes on models, Section 3.3.

```r
foo <- read.table("http://www.stat.umn.edu/geyer/5102/data/ex6-1.txt",
    header = TRUE)
gout <- glm(y ~ x, data = foo, family = binomial, x = TRUE)
summary(gout)
```

```
##
## Call:
## glm(formula = y ~ x, family = binomial, data = foo, x = TRUE)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.89589  -0.34210  -0.09359   0.34596   1.90606
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -6.7025     2.4554  -2.730  0.00634 **
## x             0.3617     0.1295   2.792  0.00524 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 40.381  on 29  degrees of freedom
## Residual deviance: 17.666  on 28  degrees of freedom
## AIC: 21.666
##
## Number of Fisher Scoring iterations: 6
```

First we need a function to calculate the log likelihood, which is almost what we did in Section 8 of the course notes on arithmetic. The function defined there was (as it finally appeared in Section 8.9

```
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)
}
```

Here, if M is the model matrix for the GLM, which is `gout$x` in the R executed above, and

$$\theta = M\beta$$

is the linear predictor for the GLM, and y is the response vector, which is `gout$y` in the R executed above, and we denote the R function `logl` defined above as $l_1(\theta, x, n)$, then the log likelihood for the logistic regression is

$$l(\beta) = \sum_{i=1}^{n} l_1(\theta_i, y_i, 1)$$

(every y_i has a binomial distribution with sample size 1).

So here is that function. We code this using the way of global variables, evil though some consider it to be (we are using the dark side of the force).

```r
modmat <- gout$x
resp <- gout$y
logl.noderiv <- function(theta, y) logl(theta, y, n = 1, deriv = 0)$value
logl.vec <- Vectorize(logl.noderiv)
logl.reg <- function(beta) {
    stopifnot(length(beta) == ncol(modmat))
    stopifnot(is.numeric(beta))
    stopifnot(is.finite(beta))
    theta <- drop(modmat %*% beta)
    sum(logl.vec(theta, resp))
}
```

8.2 A Likelihood-Based Confidence Region

A likelihood-based confidence region for all the parameters is a level set of the log likelihood, that is, a set of the form

$$\{ \beta \in R^p : (\sup l) - l(\beta) \geq c \}$$

where

$$\sup l = \sup_{\beta \in R^p} l(\beta)$$

and where c is the critical value for the likelihood ratio test (LRT), which is half the upper α critical value for the chi-square distribution with p degrees of freedom, where p is the number of parameters (betas), here $p = 2$ but we might as well state the general case. The half is because twice the log likelihood is the test statistic for the LRT.

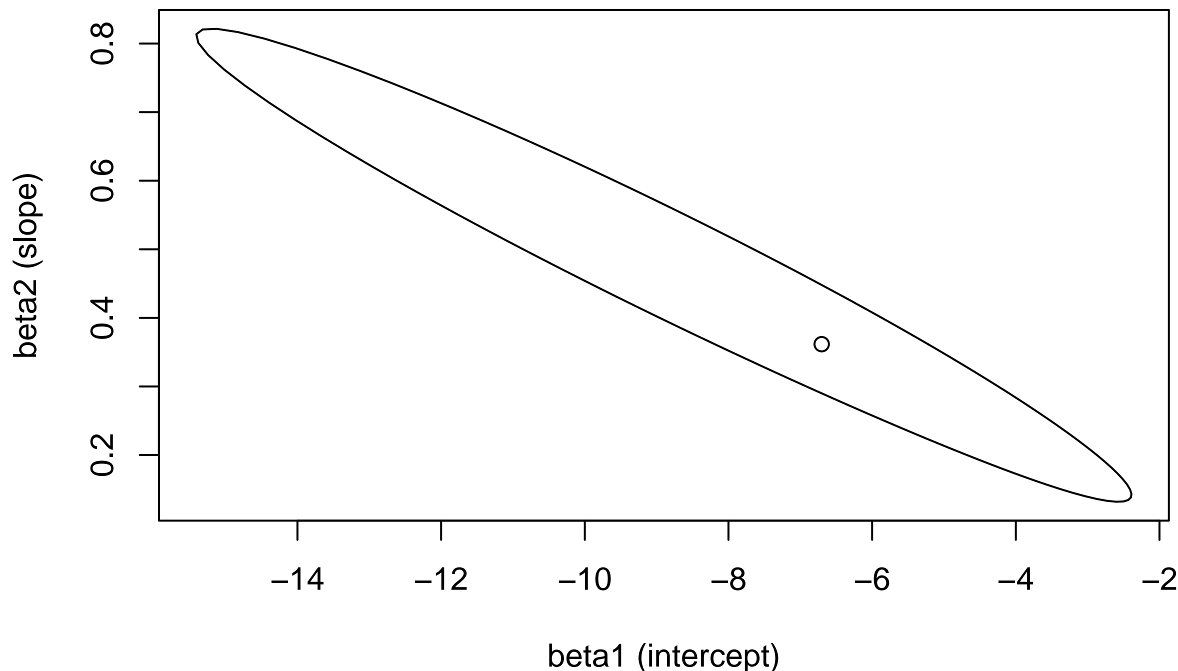
For the $p = 2$ case of our actual example the critical value is

```r
conf.level <- 0.95
alpha <- 1 - conf.level
crit <- qchisq(alpha, df = 2, lower.tail = FALSE) / 2
crit
```

```
## [1] 2.995732
```

We can draw this confidence region because this is a toy problem with $p = 2$. With more parameters, we cannot easily visualize confidence regions.

```r
beta.hat <- gout$coefficients
sup.logl.reg <- logl.reg(beta.hat)
phi <- seq(0, 2 * pi, length = 501)
rr <- double(length(phi))
for (i in seq(along = phi)) {
    fred <- function(ss) sup.logl.reg - crit -
        logl.reg(beta.hat + ss * c(20 * cos(phi[i]), sin(phi[i])))
    # 20 above because beta1 has 20 times std. err. of beta2
    rr[i] <- uniroot(fred, interval = c(0, 1), extendInt = "upX")$root
}
xx <- beta.hat[1] + rr * 20 * cos(phi)
yy <- beta.hat[2] + rr * sin(phi)
xx <- c(xx, xx[1])
yy <- c(yy, yy[1])
plot(xx, yy, type = "l", xlab = "beta1 (intercept)", ylab = "beta2 (slope)")
points(beta.hat[1], beta.hat[2])
```



This figure is very interesting. It shows clearly the dependence of the two parameter estimates (they make errors in the same direction, when one misses high the other misses low and vice versa (negative correlation)). This dependency is completely lost when one just looks at standard errors or at separate confidence intervals for each parameter.

Moreover, this figure shows clearly — if you know what to look for — that we are not in asymptopia (where "large sample theory" works perfectly). If we were in asymptopia the boundary of the confidence region would be a level set of a quadratic function that is maximized at $\hat{\beta}$, which is the dot in the figure. The curve doesn't look too different from an ellipse, but it is clearly *not* centered at the dot.

But interesting as confidence regions are, they do not scale. With three or more parameters, they exist but are not visualizable. Hence they are very rarely done. Many users of statistics have never even heard of them. Many statisticians don't even know how to make them.

8.3 Likelihood-Based Confidence Intervals

8.3.1 For Regression Coefficients

8.3.1.1 Without Derivatives

But confidence regions were not the intended subject of this section (just he warm up). We didn't need constrained optimization to do that.

Asymptotic theory also says how to make confidence regions for some but not all parameters. Suppose the parameter vector is divided into blocks $\beta = (\gamma, \delta)$. One subvector γ is the *parameters of interest* and the other subvector δ is the *nuisance parameters* (these are the actual technical terms used by theoretical statisticians). We want a confidence region for the parameter or parameters of interest that ignores the nuisance parameters. The notation suggests that the parameters of interest come before the nuisance parameters in the parameter

vector, but this is just a limitation of our notation; any parameter can be either a parameter of interest or a nuisance parameter.

Now the formula for the confidence region is

$$\left\{ \gamma \in R^q : \left(\sup_{\beta \in R^p} l(\beta) \right) - \left(\sup_{\delta \in R^{p-q}} l\big((\gamma, \delta)\big) \right) \geq c \right\}$$

where c is the critical value for the likelihood ratio test (LRT), which is half the upper α critical value for the chi-square distribution with q degrees of freedom, where q is the number of parameters of interest.

When there is only one parameter of interest we get a confidence interval. Then the critical value is

```
crit <- qchisq(alpha, df = 1, lower.tail = FALSE) / 2
crit
```

```
## [1] 1.920729
```

Of the many optimization libraries on CRAN we are going to illustrate one called `alabama` because I have tried it and it works (I tried some others that didn't work, producing nonsensical results in fairly simple problems — not every package on CRAN is great software, there is no refereeing process, there are a lot of quality control checks but they are only about whether the package follows R coding standards and will not break R, they don't check whether the code actually calculates what it claims to calculate, and the latter is particularly problematic for optimization because code can work well on some problems and fail miserably on others).

```
library(alabama)
```

```
## Loading required package: numDeriv
```

Our problem is quite simple. We want to minimize or maximize the parameter of interest, for concreteness say β_2, subject to the constraint that it lie in the level set of the log likelihood above the critical value. So our objective function is just

$$f(\beta) = \beta_2$$

but our constraint function is a complicated nonlinear function

$$l(\beta) \geq l(\hat{\beta}) - c$$

or

$$h(\beta) \geq 0$$

where

$$h(\beta) = l(\beta) - l(\hat{\beta}) + c$$

We code this up as follows.

```
m <- logl.reg(beta.hat)
h <- function(beta) logl.reg(beta) - m + crit
f <- function(beta) beta[2]
```

So now we are ready to try it out

```
aout <- auglag(beta.hat, fn = f, hin = h)
```

```
## Min(hin):  1.920729
## Outer iteration:  1
## Min(hin):  1.920729
## par:  -6.70251 0.361672
## fval =   0.3617
##
```

```
## Outer iteration:  2
## Min(hin):  0.1320565
## par:  -4.9961 0.228244
## fval =   0.2282
##
## Outer iteration:  3
## Min(hin):  0.0007093197
## par:  -3.71732 0.178068
## fval =   0.1781
##
## Outer iteration:  4
## Min(hin):  -3.868867e-05
## par:  -3.18217 0.166924
## fval =   0.1669
##
## Outer iteration:  5
## Min(hin):  -3.872968e-06
## par:  -3.18217 0.166926
## fval =   0.1669
##
## Outer iteration:  6
## Min(hin):  -3.892322e-07
## par:  -3.18217 0.166926
## fval =   0.1669
##
```

```
aout
```

```
## $par
## (Intercept)           x
##  -3.1821758   0.1669259
##
## $value
##         x
## 0.1669259
##
## $counts
## function gradient
##     1589      300
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $outer.iterations
## [1] 6
##
## $lambda
## [1] 0
##
## $sigma
## [1] 1e+07
##
```

```
## $gradient
## [1]  1.029265e-05 -8.857718e+00
##
## $ineq
## [1] -3.831501e-08
##
## $equal
## [1] NA
##
## $kkt1
##     x
## FALSE
##
## $kkt2
## [1] TRUE
##
## $hessian
##              [,1]         [,2]
## [1,]     1.840899 1.091575e+04
## [2,] 56923.331268 3.324452e+09
```

That was a bit noisy. Let's be quieter for the maximization, which it does not seem to have an option for. So we turn the objective function upside down.

```r
f <- function(beta) - beta[2]
aout.too <- auglag(beta.hat, fn = f, hin = h,
    control.outer = list(trace = FALSE))
aout.too$convergence == 0
```

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

```r
- aout.too$value
```

```
##        x
## 0.703877
```

(the minus sign is because we are minimizing minus the function we want to maximize). Thus our 95% confidence interval is the interval between our two solutions (0.167, 0.704).

8.3.1.2 Using the R function `confint` in R package `MASS`

Now we hate to tell you that R already knows how to do this calculation, so we didn't really need the code above.

```r
library(MASS)
confint(gout, "x")
```

```
## Waiting for profiling to be done...
```

```
##     2.5 %    97.5 %
## 0.1670015 0.7039358
```

But not every package that does model fitting has a `confint` function to do confidence intervals, and even if the package does have such a function, this is a useful technique to know.

8.3.1.3 With Derivatives

We are not finished. The R function `auglag` has optional arguments `gr` and `hin.jac` for derivatives of the objective function and inequality constraint function. The documentation of the former says

> The gradient of the objective function `fn` evaluated at the argument. This is a vector-function that takes a real vector as argument and returns a real vector of the same length. It defaults to `NULL`, which means that gradient is evaluated numerically. *Computations are dramatically faster in high-dimensional problems when the exact gradient is provided.*

(emphasis added). So we should provide derivatives if we can. Let's redo just the first one.

```
f <- function(beta) beta[2]
gr <- function(beta) c(0, 1)
```

The documentation is a bit unclear as to exactly what form the Jacobian (matrix of partial derivatives) of the inequality constraint function is supposed to take (is it a vector or a matrix). But when we didn't supply it, it was calculated by the `numDeriv` package, so we can see what form it should have by doing an example

```
grad(h, beta.hat)
```

```
## [1] -2.392129e-11  1.365607e-10
```

Looks like it can be a vector.

```
logl.deriv <- function(theta, y) logl(theta, y, n = 1, deriv = 1)$gradient
logl.deriv.vec <- Vectorize(logl.deriv)
logl.reg.deriv <- function(beta) {
    stopifnot(length(beta) == ncol(modmat))
    stopifnot(is.numeric(beta))
    stopifnot(is.finite(beta))
    theta <- drop(modmat %*% beta)
    dtheta <- logl.deriv.vec(theta, resp)
    ### now apply multivariable chain rule
    drop(dtheta %*% modmat)
}
beta.try <- beta.hat + rnorm(2) * 0.1
logl.reg.deriv(beta.try)
```

```
## (Intercept)           x
##    1.483556   30.796135
```

```
logl.reg.deriv(beta.hat)
```

```
##   (Intercept)             x
## -2.984134e-11  1.959247e-10
```

(I have to admit the code above is not exactly obvious. It took me a while to get right.)

```
aout <- auglag(beta.hat, fn = f, gr = gr, hin = h, hin.jac = logl.reg.deriv,
    control.outer = list(trace = FALSE))
```

```
## Error in hin.jac(par, ...)[active, , drop = FALSE]: incorrect number of dimensions
```

Looks like my guess as to what shape the result of `hin.jac` was supposed to be was wrong. Try two.

```
logl.reg.deriv <- function(beta) {
    stopifnot(length(beta) == ncol(modmat))
    stopifnot(is.numeric(beta))
    stopifnot(is.finite(beta))
    theta <- drop(modmat %*% beta)
    dtheta <- logl.deriv.vec(theta, resp)
    ### now apply multivariable chain rule
    dtheta %*% modmat
```

```
}
logl.reg.deriv(beta.try)
```

```
##      (Intercept)        x
## [1,]    1.483556 30.79613
```

```
logl.reg.deriv(beta.hat)
```

```
##        (Intercept)            x
## [1,] -2.984134e-11 1.959247e-10
```

Now it returns a $1 \times p$ matrix. Does that work?

```
aout <- auglag(beta.hat, fn = f, gr = gr, hin = h, hin.jac = logl.reg.deriv,
    control.outer = list(trace = FALSE))
aout$convergence == 0
```

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

```
aout$value
```

```
##         x
## 0.1669259
```

Success!

8.3.2 For Mean Value Parameters

But that still wasn't the problem we really wanted to do. Let's do something that `confint.glm` cannot do. Let's make the likelihood based competitor of the prediction interval we did in Section 3.3.4 of the course notes about models. There, recall we were trying to get a confidence interval for the mean value (under the true unknown value for β) for the predictor value $x = 25$.

So now our objective function and its derivative function are

```
f <- function(beta) beta[1] + 25 * beta[2]
gr <- function(beta) c(1, 25)
```

(this is for the linear predictor, we will have to map our results through the inverse logit function to get the mean value parameter, as we did in the notes linked above — the reason for doing this in two steps is that the derivative is so much simpler for the linear predictor).

```
aout <- auglag(beta.hat, fn = f, gr = gr, hin = h, hin.jac = logl.reg.deriv,
    control.outer = list(trace = FALSE))
aout$convergence == 0
```

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

```
aout$value
```

```
## (Intercept)
##   0.6510069
```

```
f.too <- function(beta) - (beta[1] + 25 * beta[2])
gr.too <- function(beta) - c(1, 25)

aout.too <- auglag(beta.hat, fn = f.too, gr = gr.too,
    hin = h, hin.jac = logl.reg.deriv,
    control.outer = list(trace = FALSE))
aout.too$convergence == 0
```

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

```r
- aout.too$value
```

```
## (Intercept)
##    4.990683
```

```r
invlogit <- function(theta) 1 / (1 + exp(- theta))
invlogit(c(aout$value, - aout.too$value))
```

```
## (Intercept) (Intercept)
##   0.6572373   0.9932449
```

For comparison, we repeat the other confidence intervals we got in the course notes cited above

```r
zcrit <- qnorm((1 + conf.level) / 2)
pout <- predict(gout, newdata = data.frame(x = 25),
    se.fit = TRUE, type = "response")
pout$fit + c(-1,1) * zcrit * pout$se.fit
```

```
## [1] 0.7468731 1.0772870
```

```r
pout.too <- predict(gout, newdata = data.frame(x = 25),
    se.fit = TRUE)
pout.too$fit + c(-1,1) * zcrit * pout.too$se.fit
```

```
## [1] 0.2791077 4.3994944
```

```r
invlogit(pout.too$fit + c(-1,1) * zcrit * pout.too$se.fit)
```

```
## [1] 0.5693274 0.9878655
```

Note that these are confidence intervals for different parameters (θ and μ).

To summarize, our 95% confidence intervals for θ for $x = 25$ were

- likelihood based: (0.651, 4.991)
- default (a. k. a., Wald — estimate plus or minus 1.96 standard errors): (0.279, 4.399)

and our 95% confidence intervals for μ for $x = 25$ were

- likelihood based: (0.657, 0.993)
- Wald: (0.747, 1.077)
- Wald for θ mapped to μ: (0.569, 0.988)

All of these confidence intervals (for the same parameter) are asymptotically equivalent. They are equal in asymptopia (when the amount of data has gone all the way to infinity). And they are nearly equal when the data are very, very large. Here we are clearly not in asymptopia (as we already saw in the course notes linked above from the disagreement of the last two intervals). Now we have an even better interval, and one that no pre-existing R function computes.

9 Summary

- If you know about optimization, you can do a lot of things that haven't already been pre-packaged for you. You aren't limited to cookbook statistics.
- Frequentist statistics has its own sort of TIMTOWTDI. Any statistic you call a point estimator of a parameter *is* a point estimator of that parameter. (Of course, some are better than others.) Any pair of statistics you say are endpoints of a confidence interval for a parameter *do* define a confidence interval for that parameter. (Of course, some are better than others.)