Advanced Statistical Computing

Roger D. Peng

Advanced Statistical Computing

Roger D. Peng

2018-07-17

Contents

Welcome

The book covers material taught in the Johns Hopkins Biostatistics Advanced Statistical Computing course. I taught this course off and on from 2003–2016 to upper level PhD students in Biostatistics. The course ran for 8 weeks each year, which is a fairly compressed schedule for material of this nature. Because of the short time frame, I felt the need to present material in a manner that assumed that students would often be using others' software to implement these algorithms but that they would need to know what was going on underneath. In particular, should something go wrong with one of these algorithms, it's important that they know enough to diagnose the problem and make an attempt at fixing it. Therefore, the book is a bit light on the details, in favor of giving a more general overview of the methods.

This book is a WORK IN PROGRESS.

Stay in Touch!

If you are interested in hearing more from me about things that I'm working on (books, data science courses, podcast, etc.), you can do two things:

- First, I encourage you to join the Johns Hopkins Data Science Lab mailing list. On this list I send out updates of my own activities as well as occasional comments on data science current events. You can also find out what my co-conspirators Jeff Leek, Brian Caffo, and Stephanie Hicks are up to because sometimes they do really cool stuff.
- Second, I have a regular podcast called Not So Standard Deviations that I co-host with Dr. Hilary Parker, a Data Scientist at Stitch Fix. On this podcast, Hilary and I talk about the craft of data science and discuss common issues and problems in analyzing data. We also compare how data science is approached in both academia and industry contexts and discuss the latest industry trends. You can listen to recent episodes on our web site or you can subscribe to it in Apple Podcasts or your favorite podcasting app.

For those of you who purchased a **printed copy** of this book, I encourage you to go to the Leanpub web site and obtain the e-book version, which is available for free. The reason is that I will occasionally update the book with new material and readers who purchase the e-book version are entitled to free updates (this is unfortunately not yet possible with printed books) and will be notified when they are released. You can also find a web version of this book at the bookdown web site.

Thanks again for purchasing this book and please do stay in touch!

Setup

This book makes use of the following R packages, which should be installed to take full advantage of the examples.

```
dplyr
ggplot2
knitr
MASS
microbenchmark
mvtnorm
readr
remotes
tidypvals
tidyr
```

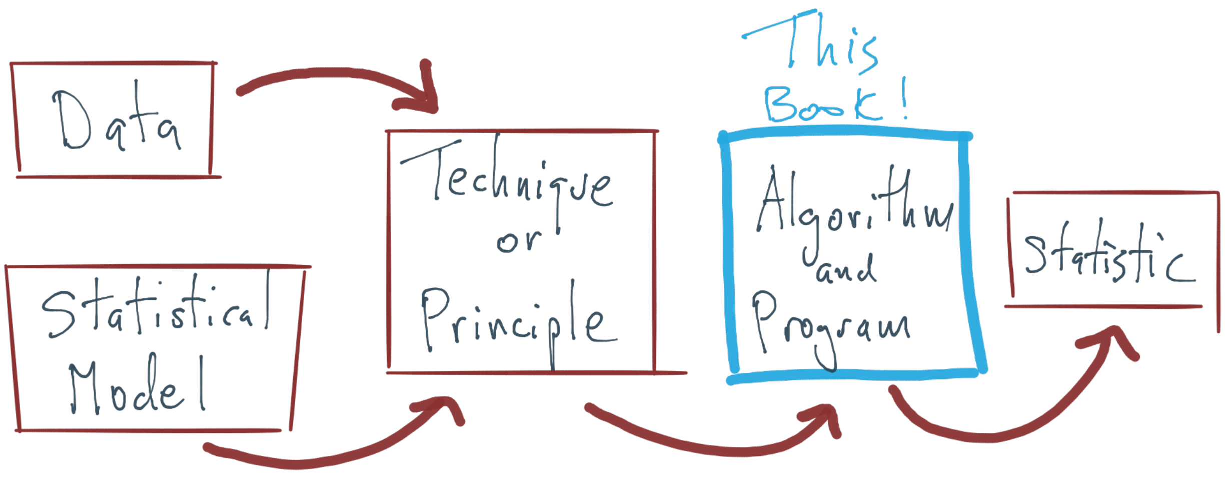


Figure 1: The process of statistical modeling.

1 Introduction

The journey from statistical model to useful output has many steps, most of which are taught in other books and courses. The purpose of this book is to focus on one particular aspect of this journey: the development and implementation of statistical algorithms.

It's often nice to think about statistical models and various inferential philosophies and techniques, but when the rubber meets the road, we need an algorithm and a computer program implementation to get the results we need from a combination of our data and our models. This book is about how we fit models to data and the algorithms that we use to do so.

1.1 Example: Linear Models

Consider the simple linear model.

$$y = \beta_0 + \beta_1 x + \varepsilon \tag{1}$$

This model has unknown parameters β_0 and β_1. Given observations $(y_1, x_1), (y_2, x_2), \dots, (y_n, x_n)$, we can combine these data with the *likelihood principle*, which gives us a procedure for producing model parameter estimates. The likelihood can be *maximized* to produce *maximum likelihood estimates*,

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

and

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})}$$

These *statistics*, $\hat{\beta}_0$ and $\hat{\beta}_1$, can then be interpreted, depending on the area of application, or used for other purposes, perhaps as inputs to other procedures. In this simple example, we can see how each component of the modeling process works.

Component	Implementation
Model	Linear regression
Principle/Technique	Likelihood principle

Component	Implementation
Algorithm	Maximization
Statistic	$\hat{\beta}_0$, $\hat{\beta}_1$

In this example, the maximization of the likelihood was simple because the solution was available in closed form. However, in most other cases, there will not be a closed form solution and some specific algorithm will be needed to maximize the likelihood.

Changing the implementation of a given component can lead to different outcomes further down the change and can even produce completely different outputs. Identical estimates for the parameters in this model can be produced (in this case) by replacing the likelihood principle with the principle of least squares. However, changing the principle to produce, for example, maximum *a posteriori* estimates would have produced different statistics at the end.

1.2 Principle of Optimization Transfer

There is a general principle that will be repeated in this book that Kenneth Lange calls "optimization transfer". The basic idea applies to the problem of maximizing a function f.

1. We want to maximize f, but it is difficult to do so.
2. We *can* compute an approximation to f, call it g, based on local information about f.
3. Instead of maximizing f, we "transfer" the maximization problem to g and maximize g instead.
4. We iterate Steps 2 and 3 until convergence.

The difficult problem of maximizing f is replaced with the simpler problem of maximizing g coupled with *iteration* (otherwise known as computation). This is the optimization "transfer".

Note that all of the above applies to minimization problems, because maximizing f is equivalent to minimizing $-f$.

1.3 Textbooks vs. Computers

One confusing aspect of statistical computing is that often there is a disconnect between what is printed in a statistical computing textbook and what *should* be implemented on the computer. In textbooks, it is usually simpler to present solutions as convenient mathematical formulas whenever possible, in order to communicate basic ideas and to provide some insight. However, directly translating these formulas into computer code is usually not advisable because there are many problematic aspects of computers that are simply not relevant when writing things down on paper.

Some key issues to look for when implementing statistical or numerical solutions on the computer are

1. Overflow - When numbers get too big, they cannot be represented on a computer and so often `NA`s are produced instead;
2. Underflow - Similar to overflow, numbers can get too small for computers to represent, resulting in errors or warnings or inaccurate computation;
3. Near linear dependence - the existence of linear dependence in matrix computations depends on the precision of a machine. Because computers are finite precision, there are commonly situations where one might think there is no linear dependence but the computer cannot tell the difference.

All three of the above problems arise from the finite precision nature of all computers. One must take care to use algorithms that do calculations in the computable range and that automatically handle things like near dependence.

Below, I highlight some common examples in statistics where the implementation diverges from what textbooks explain as the solution: Computing with logarithms, the least squares solution to the linear regression estimation problem, and the computation of the multivariate Normal density. Both problems, on paper, involve inverting a matrix, which is typically a warning sign in any linear algebra problem. While matrix inverses are commonly found in statistics textbooks, it's rare in practice that you will ever want to directly compute them. This point bears repeating: **If you find yourself computing the inverse of a matrix, there is usually a better way of doing whatever you are trying to do**.

1.3.1 Using Logarithms

Most textbooks write out functions, such as densities, in their natural form. For example, the univariate Normal distribution with mean μ and variance σ^2 is written

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

and you can compute this value for any x, μ, and σ in R with the `dnorm()` function.

But in practice, you almost never have to compute this exact number. Usually you can get away with computing the log of this value (and with `dnorm()` you can set the option `log = TRUE`). In some situations, such as with importance sampling, you do have to compute density values on the original scale, and that can be considered a disadvantage of that technique.

Computing densities with logarithms is much more numerically stable than computing densities without them. With the exponential function in the density, numbers can get very small quickly, to the point where they are too small for the machine to represent (underflow). In some situations, you may need to take the ratio of two densities and then you may end up with either underflow or overflow (if numbers get too big). Doing calculations on a log scale (and then exponentiating them later if needed) usually resolves problems of underflow or overflow.

In this book (and in any other), when you see expressions like $f(x)/g(x)$, you should think that this means $\exp(\log f(x) - \log(g(x)))$. The two are equivalent but the latter is likely more numerically stable. In fact, most of the time, you never have to re-exponentiate the values, in which case you can spend your entire time in log-land. For example, in the rejection sampling algorithm, you need to determine if $U \leq \frac{f(x)}{g(x)}$. However, taking the log of both sides allows you to do the exact same comparison in a much more numerically stable way.

1.3.2 Linear Regression

The typical linear regression model, written in matrix form, is represented as follows,

$$y = X\beta + \varepsilon$$

where y is an $n \times 1$ observed response, X is the $n \times p$ predictor matrix, β is the $p \times 1$ coefficient vector, and ε is $n \times 1$ error vector.

In most textbooks the solution for estimating β, whether it be via maximum likelihood or least squares, is written as

$$\hat{\beta} = (X'X)^{-1}X'y.$$

And indeed, that *is* the solution. In R, this could be translated literally as

```
betahat <- solve(t(X) %*% X) %*% t(X) %*% y
```

where `solve()` is used to invert the cross product matrix $X'X$. However, one would never compute the actual value of $\hat{\beta}$ this way on the computer. The formula presented above is *only* computed in textbooks.

The primary reason is that computing the direct inverse of $X'X$ is very expensive computationally and is a potentially unstable operation on a computer when there is high colinearity amongst the predictors. Furthermore, in computing $\hat{\beta}$ we do not actually need the inverse of $X'X$, so why compute it? A simpler approach would be to take the normal equations,

$$X'X\beta = X'y$$

and solve them directly. In R, we could write

```
solve(crossprod(X), crossprod(X, y))
```

Rather than compute the inverse of $X'X$, we directly compute $\hat{\beta}$ via Gaussian elimination. This approach has the benefit of being more numerically stable and being *much* faster.

```
set.seed(2017-07-13)
X <- matrix(rnorm(5000 * 100), 5000, 100)
y <- rnorm(5000)
```

Here we benchmark the naive computation.

```
library(microbenchmark)
microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y)
```

```
Unit: milliseconds
                             expr      min       lq     mean   median
 solve(t(X) %*% X) %*% t(X) %*% y 8.797343 9.835521 11.94031 10.85574
       uq      max neval
 13.02197 57.42063   100
```

The following timing uses the `solve()` function to compute $\hat{\beta}$ via Gaussian elimination.

```
microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y,
               solve(crossprod(X), crossprod(X, y)))
```

```
Unit: milliseconds
                                 expr      min       lq      mean   median
     solve(t(X) %*% X) %*% t(X) %*% y 8.187201 8.957787 10.140894 9.197867
 solve(crossprod(X), crossprod(X, y)) 1.841551 1.934159  2.012302 1.992533
        uq       max neval
 11.096826 36.599512   100
  2.061172  3.284126   100
```

You can see that the betweeen the two approach there is a more than 5-fold difference in computation time, with the second approach being considerably faster.

However, this approach breaks down when there is any colinearity in the X matrix. For example, we can tack on a column to X that is very similar (but not identical) to the first column of X.

```
W <- cbind(X, X[, 1] + rnorm(5000, sd = 0.0000000001))
solve(crossprod(W), crossprod(W, y))
```

```
Error in solve.default(crossprod(W), crossprod(W, y)): system is computationally singular: reciprocal c
```

Now the approach doesn't work because the cross product matrix $W'W$ is singular. In practice, matrices like these can come up a lot in data analysis and it would be useful to have a way to deal with it automatically.

R takes a different approach to solving for the unknown coefficients in a linear model. R uses the QR decomposition, which is not as fast, but has the added benefit of being able to automatically detect and handle colinear columns in the matrix.

Here, we use the fact that X can be decomposed as $X = QR$, where Q is an orthonormal matrix and R is an upper triangular matrix. Given that, we can write

$$X'X\beta = X'y$$

as

$$\begin{aligned} R'Q'QR\beta &= R'Q'y \\ R'R\beta &= R'Q'y \\ R\beta &= Q'y \end{aligned}$$

because $Q'Q = I$. At this point, we can solve for β via Gaussian elimination, which is greatly simplified because R is already upper triangular. The QR decomposition has the added benefit that we do not have to compute the cross product $X'X$ at all, as this matrix can be numericaly unstable if it is not properly centered or scaled.

We can see in R code that even with our singular matrix `W` above, the QR decomposition continues without error.

```r
Qw <- qr(W)
str(Qw)
```

```
List of 4
 $ qr   : num [1:5000, 1:101] 70.88664 -0.01277 0.01561 0.00158 0.02451 ...
 $ rank : int 100
 $ qraux: num [1:101] 1.01 1.03 1.01 1.02 1.02 ...
 $ pivot: int [1:101] 1 2 3 4 5 6 7 8 9 10 ...
 - attr(*, "class")= chr "qr"
```

Note that the output of `qr()` computes the rank of W to be 100, not 101, because of the colinear column. From there, we can get $\hat{\beta}$ if we want using `qr.coef()`,

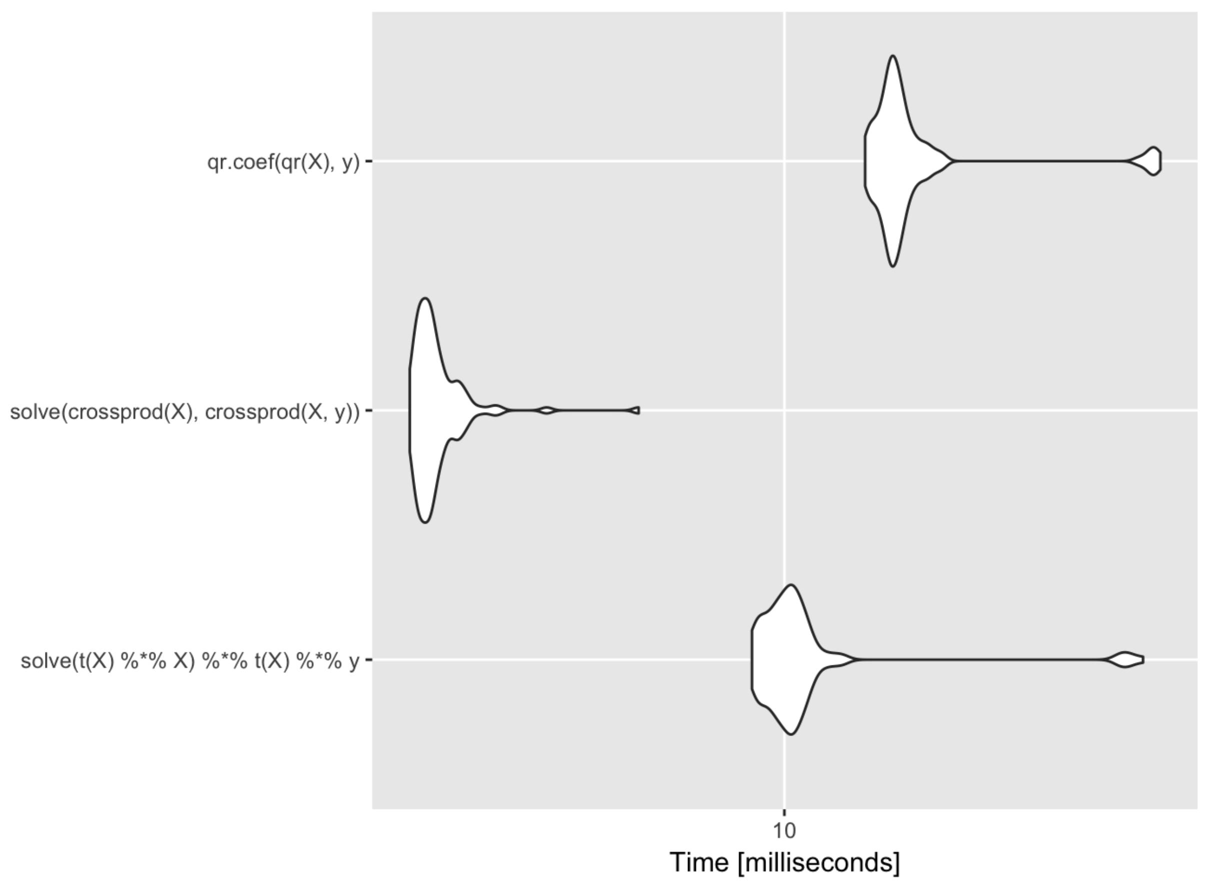
```r
betahat <- qr.coef(Qw, y)
```

We do not show it here, but the very last element of `betahat` is `NA` because a coefficient corresponding to the last column of W (the collinear column) could not be calculated.

While the QR decomposition does handle colinearity, we do pay a price in speed.

```r
library(ggplot2)
m <- microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y,
                    solve(crossprod(X), crossprod(X, y)),
                    qr.coef(qr(X), y))
autoplot(m)
```

```
Coordinate system already present. Adding new coordinate system, which will replace the existing one.
```



Compared to the approaches above, it is comparable to the naive approach but it is a much better and more stable method.

In practice, we do not use functions like `qr()` or `qr.coef()` directly because higher level functions like `lm()` do the work for us. However, for certain narrow, highly optimized cases, it may be fruitful to turn to another matrix decomposition to compute linear regression coefficients, particularly if this must be done repeatedly in a loop.

1.3.3 Multivariate Normal Distribution

Computing the multivariate normal density is a common problem in statistics, such as in fitting spatial statistical models or Gaussian process models. Because optimization procedures used to compute maximum likelihood estimates or likelihood ratios can be evaluated hundreds or thousands of times in a single run, it's useful to have a highly efficient procedure for evaluating the multivariate Normal density.

The p-dimensional multivariate Normal density is written as

$$\varphi(x \mid \mu, \Sigma) = -\frac{p}{2}\log 2\pi - \frac{1}{2}\log|\Sigma| - \frac{1}{2}(x-\mu)'\Sigma^{-1}(x-\mu)$$

The critical, and most time-consuming, part of computing the multivariate Normal density is the quadratic form,

$$(x-\mu)'\Sigma^{-1}(x-\mu).$$

We can simplify this problem a bit by focusing on the centered version of x which we will refer to as $z = x - \mu$. Hence, we are trying to compute

$$z'\Sigma^{-1}z$$

Here, much like the linear regression example above, the key bottleneck is the inversion of the p-dimensional covariance matrix Σ. If we take z to be a $p \times 1$ column vector, then a literal translation of the mathematics into R code might look something like this,

```
t(z) %*% solve(Sigma) %*% z
```

But once again, we are taking on the difficult and unstable task of inverting Σ when, at the end of the day, we do not need this inverse.

Instead of taking the textbook translation approach, we can make use of the Cholesky decomposition of Σ. The Cholesky decomposition of a positive definite matrix provides

$$\Sigma = R'R$$

where R is an upper triangular matrix. R is sometimes referred to as the "square root" of Σ (although it is not unique). Using the Cholesky decomposition of Σ and the rules of matrix algebra, we can then write

$$\begin{aligned} z'\Sigma^{-1}z &= z'(R'R)^{-1}z \\ &= z'R^{-1}R'^{-1}z \\ &= (R'^{-1}z)'R'^{-1}z \\ &= v'v \end{aligned}$$

where $v = R'^{-1}z$ and is a $p \times 1$ vector. Furthermore, we can avoid inverting R' by computing v as the solution to the linear system

$$R'v = z$$

Once we have computed v, we can compute the quadratic form as $v'v$, which is simply the cross product of two p-dimensional vectors!

Another benefit of the Cholesky decomposition is that it gives us a simple way to compute the log-determinant of Σ. The log-determinant of Σ is simply 2 times the sum of the log of the diagonal elements of R.

Here is an implementation of the naive approach to computing the quadratic form in the multivariate Normal.

```
set.seed(2017-07-13)
z <- matrix(rnorm(200 * 100), 200, 100)
S <- cov(z)
quad.naive <- function(z, S) {
        Sinv <- solve(S)
        rowSums((z %*% Sinv) * z)
}
```

We can first take a look at the output that this function produces.

```
library(dplyr)
quad.naive(z, S) %>% summary
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  73.57   93.54  100.59  100.34  106.39  129.12 
```

The following is a version of the quadratic form function that uses the Cholesky decomposition.

```
quad.chol <- function(z, S) {
        R <- chol(S)
        v <- backsolve(R, t(z), transpose = TRUE)
        colSums(v * v)
}
```

We can verify that this function produces the same output as the naive version.

```
quad.chol(z, S) %>% summary
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  73.57   93.54  100.59  100.34  106.39  129.12 
```

Now, we can time both procedures to see how they perform.

```
library(microbenchmark)
microbenchmark(quad.naive(z, S), quad.chol(z, S))
```

```
Unit: microseconds
             expr     min        lq      mean   median       uq       max
 quad.naive(z, S) 593.125 749.9505 1075.3927 830.0240 903.1380 17847.149
  quad.chol(z, S) 252.818 448.3040  592.0377 485.8535 528.9575  9492.268
 neval
   100
   100
```

We can see that the version using the Cholesky decomposition takes about 60% of the time of the naive version. In a single evaluation, this may not amount to much time. However, over the course of potentially many iterations, these kinds of small savings can add up.

The key lesson here is that our use of the Cholesky decomposition takes advantage of the fact that we know that the covariance matrix in a multivariate Normal is symmetric and positive definite. The naive version of the algorithm that just blindly inverts the covariance matrix is not able to take advantage of this information.

2 Solving Nonlinear Equations

2.1 Bisection Algorithm

The bisection algorithm is a simple method for finding the roots of one-dimensional functions. The goal is to find a root $x_0 \in [a, b]$ such that $f(x_0) = 0$. The algorithm starts with a large interval, known to contain x_0, and then successively reduces the size of the interval until it brackets the root. The theoretical underpinning of the algorithm is the intermediate value theorem which states that if a continuous function f takes values $f(a)$ and $f(b)$ at the end points of the interval $[a, b]$, then f must take all values between $f(a)$ and $f(b)$ somewhere in the interval. So if $f(a) < \gamma < f(b)$, then there exists a $c \in [a, b]$ such that $f(c) = \gamma$.

Using this information, we can present the bisection algorithm. First we must check that $\text{sign}(f(a)) \neq \text{sign}(f(b))$. Otherwise, the interval does not contain the root and might need to be widened. Then we can proceed:

1. Let $c = \frac{a+b}{2}$.
2. If $f(c) = 0$, stop and return c.
3. If $\text{sign}(f(a)) \neq \text{sign}(f(c))$, then set $b \leftarrow c$. Else if $\text{sign}(f(b)) \neq \text{sign}(f(c))$, then set $a \leftarrow c$.
4. Goto the beginning and repeat until convergence (see below).

After n iterations, the size of the interval bracketing the root will be $2^{-n}(b - a)$.

The bisection algorithm is useful, conceptually simple, and is easy to implement. In particular, you do not need any special information about the function f except the ability to evaluate it at various points in the interval. The downsides are that it is only useful in one dimension and its convergence is linear, which is the slowest rate of convergence for algorithms we will discuss (more on that later).

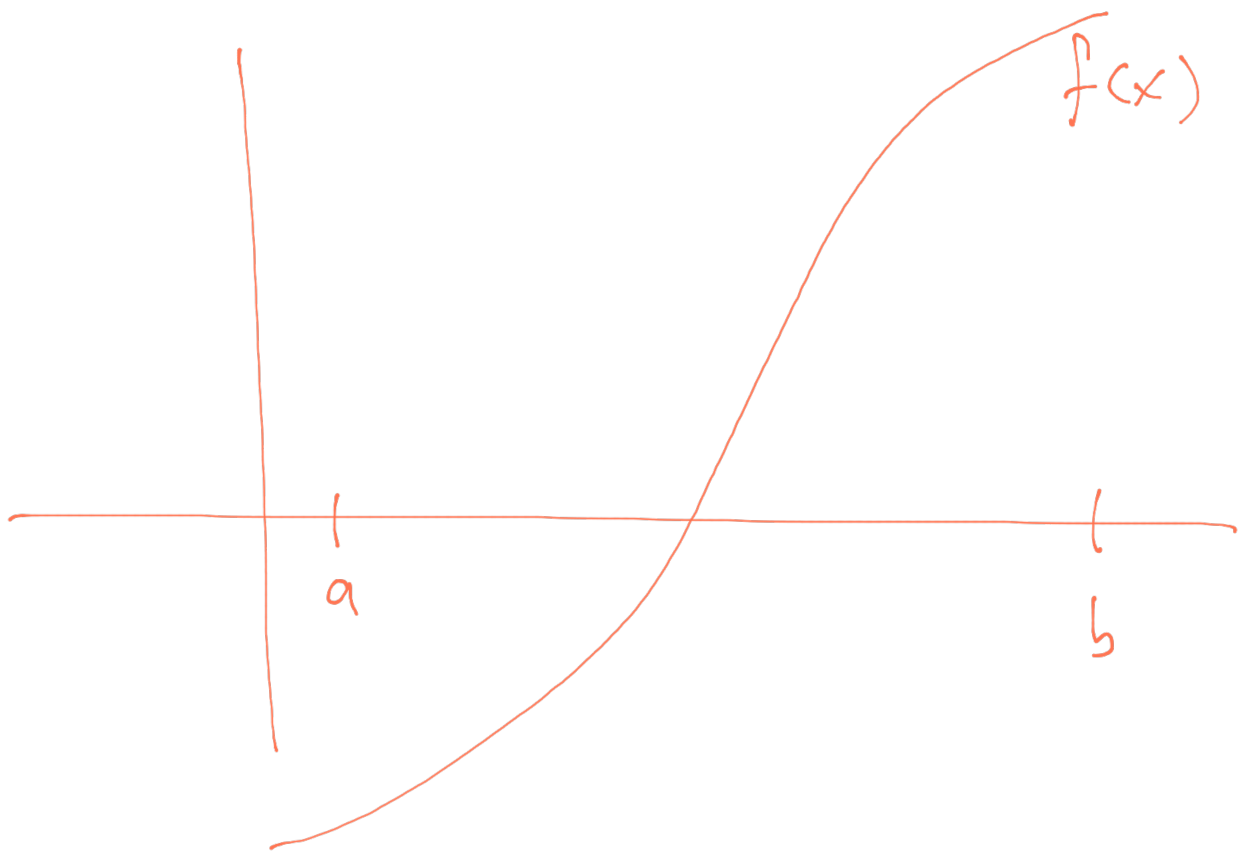


Figure 2: Ideal setup for bisection algorithm.

The bisection algorithm can run into problems in situations where the function f is not well behaved. The ideal situation for the bisection algorithm looks something like this.

Here, $f(a)$ and $f(b)$ are of opposite signs and the root is clearly in between a and b.

In the scenario below, the algorithm will not start because $f(a) > 0$ and $f(b) > 0$.

In this next scenario, there are two roots between a and b, in addition to having $f(a) > 0$ and $f(b) > 0$. One would need to reduce the length of the starting interval in order to find either root.

In the scenario below, the algorithm will start because $f(a)$ and $f(b)$ are of opposite sign, but there is no root.

Convergence of the bisection algorithm can be determined by either having $|b - a| < \varepsilon$ for some small ε or having $|f(b) - f(a)| < \varepsilon$. Which criterion you use will depend on the specific application and on what kinds of tolerances are required.

2.1.1 Example: Quantiles

Given a cumulative distribution function $F(x)$ and a number $p \in (0, 1)$, a quantile of F is a number x such that $F(x) = p$. The bisection algorithm can be used to find a quantile x for a given p by defining the function $g(x) = F(x) - p$ and solving for the value of x that achieves $g(x) = 0$.

Another way to put this is that we are inverting the CDF to compute $x = F^{-1}(p)$. So the bisection algorithm can be used to invert functions in these situations.

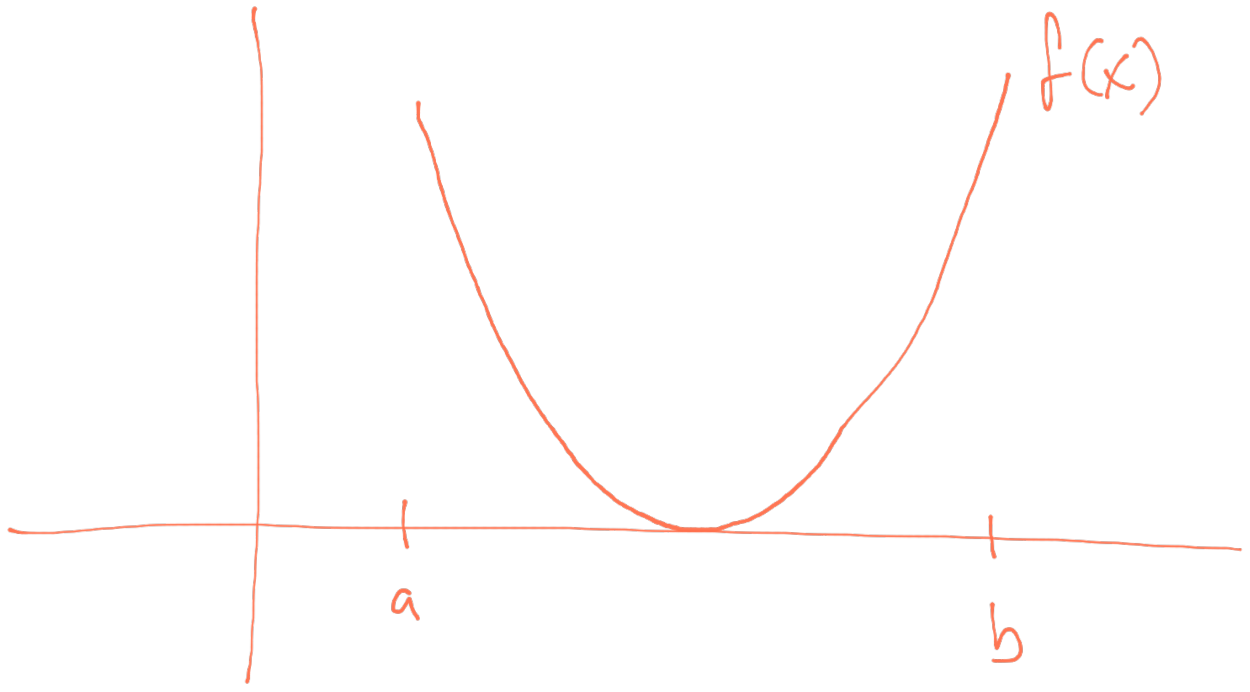


Figure 3: Derivative of f at the root is 0.

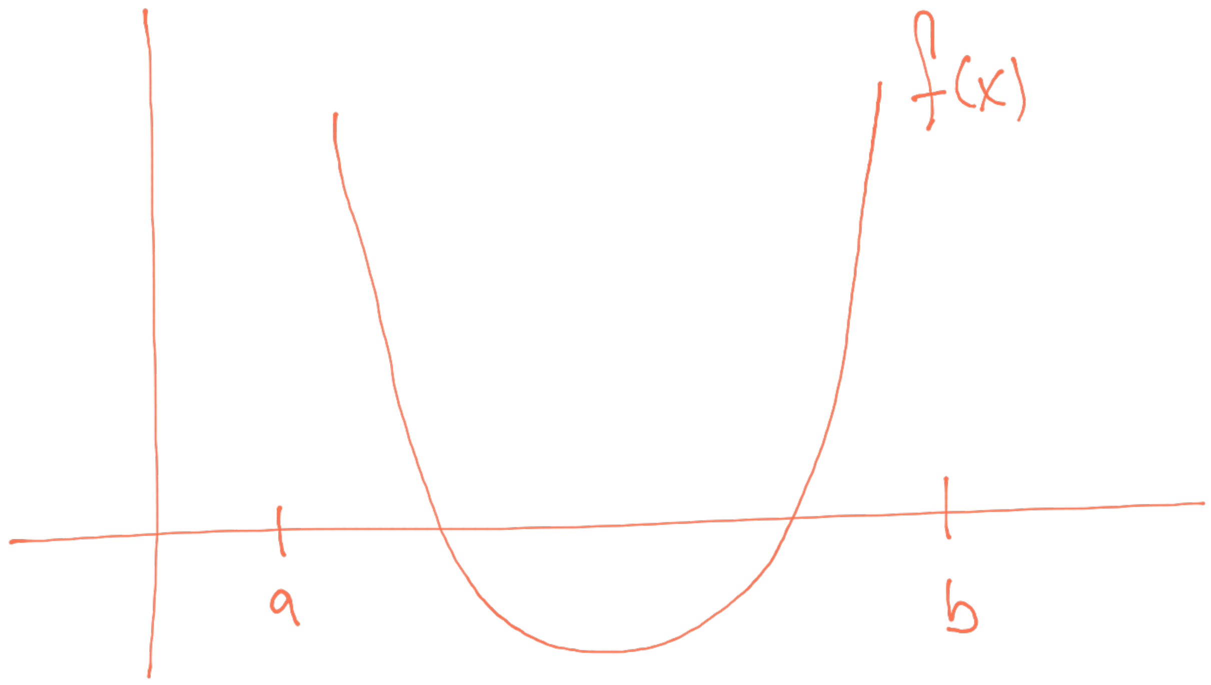


Figure 4: Two roots within the interval.

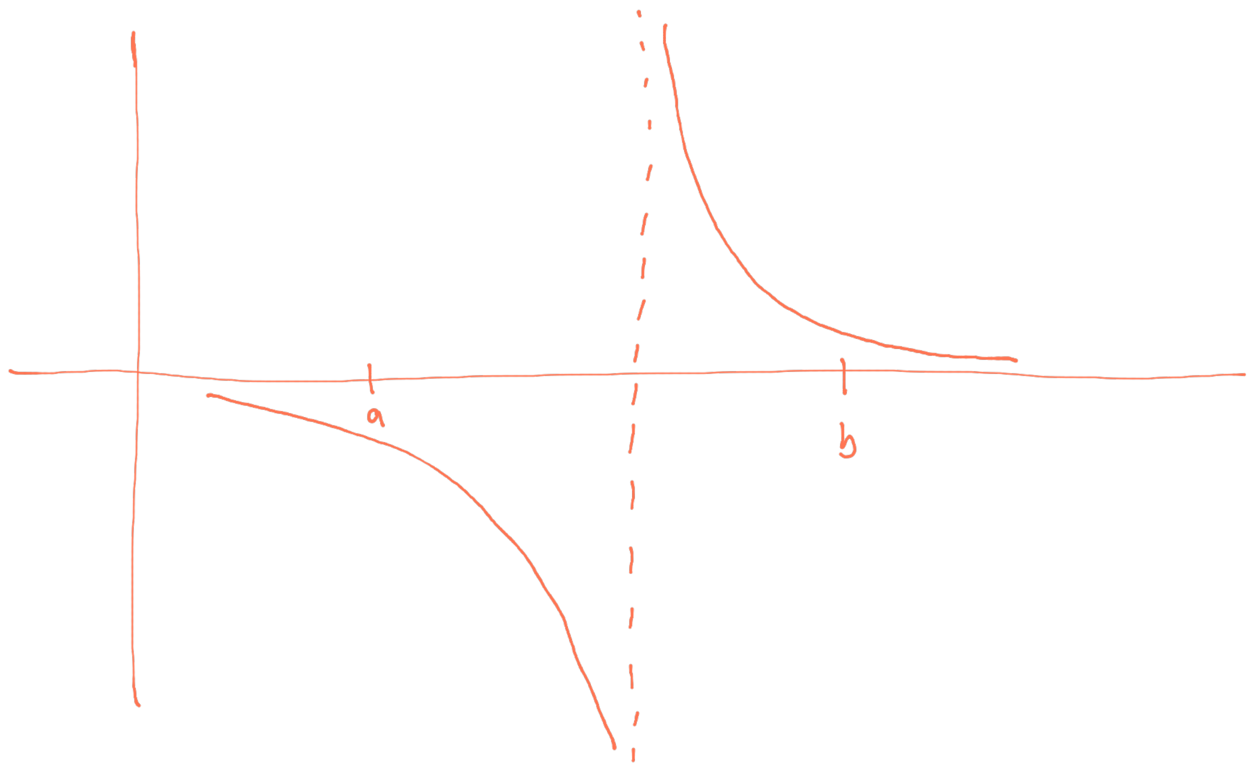


Figure 5: Interval contains an asymptote but no root.

2.2 Rates of Convergence

One of the ways in which algorithms will be compared is via their rates of convergence to some limiting value. Typically, we have an interative algorithm that is trying to find the maximum/minimum of a function and we want an estimate of how long it will take to reach that optimal value. There are three rates of convergence that we will focus on here—linear, superlinear, and quadratic—which are ordered from slowest to fastest.

In our context, rates of convergence are typically determined by how much information about the target function f we use in the updating process of the algorithm. Algorithms that use little information about f, such as the bisection algorithm, converge slowly. Algorithms that require more information about f, such as derivative information, typically converge more quickly. There is no free lunch!

2.2.1 Linear convergence

Suppose we have a sequence $\{x_n\}$ such that $x_n \to x_\infty$ in $\mathfrak{R}^k$. We say the convergence is *linear* if there exists $r \in (0, 1)$ such that

$$\frac{\|x_{n+1} - x_\infty\|}{\|x_n - x_\infty\|} \leq r$$

for all n sufficiently large.

2.2.1.1 Example

The simple sequence $x_n = 1 + \left(\frac{1}{2}\right)^n$ converges linearly to $x_\infty = 1$ because

$$\frac{\|x_{n+1}-x_\infty\|}{\|x_n-x_\infty\|} = \frac{\left(\frac{1}{2}\right)^{n+1}}{\left(\frac{1}{2}\right)^{n}} = \frac{1}{2}$$

which is always in $(0, 1)$.

2.2.2 Superlinear Convergence

We say a sequence $\{x_n\}$ converges to x_∞ *superlinearly* if we have

$$\lim_{n\to\infty} \frac{\|x_{n+1}-x_\infty\|}{\|x_n-x_\infty\|} = 0$$

The sequence above does not converge superlinearly because the ratio is always constant, and so never can converge to zero as $n \to \infty$. However, the sequence $x_n = 1 + \left(\frac{1}{n}\right)^n$ converges superlinearly to 1.

2.2.3 Quadratic Convergence

Quadratic convergence is the fastest form of convergence that we will discuss here and is generally considered desirable if possible to achieve. We say the sequence converges at a *quadratic* rate if there exists some constant $0 < M < \infty$ such that

$$\frac{\|x_{n+1}-x_\infty\|}{\|x_n-x_\infty\|^2} \leq M$$

for all n sufficiently large.

Extending the examples from above, the sequence $x_n = 1 + \left(\frac{1}{n}\right)^{2^n}$ converges quadratically to 1. With this sequence, we have

$$\frac{\|x_{n+1}-x_\infty\|}{\|x_n-x_\infty\|^2} = \frac{\left(\frac{1}{n+1}\right)^{2^{n+1}}}{\left(\frac{1}{n}\right)^{(2^n)2}} = \left(\frac{n}{n+1}\right)^{2^{n+1}} \leq 1$$

2.2.4 Example: Bisection Algorithm

For the bisection algorithm, the error that we make in estimating the root is $x_n = |b_n - a_n|$, where a_n and b_n represent the end points of the bracketing interval at iteration n. However, we know that the size of the interval in the bisection algorithm decreases by a half at each iteration. Therefore, we can write $x_n = 2^{-n}|b_0 - a_0|$ and we can write the rate of convergence as

$$\frac{\|x_{n+1}-x_\infty\|}{\|x_n-x_\infty\|} = \frac{x_{n+1}}{x_n} = \frac{2^{-(n+1)}(b_0-a_0)}{2^{-n}(b_0-a_0)} = \frac{1}{2}$$

Therefore, the error of the bisection algorithm converges linearly to 0.

2.3 Functional Iteration

We want to find a solution to the equation $f(x) = 0$ for $f : \mathbb{R}^k \to \mathbb{R}$ and $x \in S \subset \mathbb{R}^k$. One approach to solving this problem is to characterize solutions as *fixed points* of other functions. For example, if $f(x_0) = 0$, then x_0 is a fixed point of the function $g(x) = f(x) + x$. Another such function might be $g(x) = x(f(x) + 1)$ for $x \neq 0$.

In some situations, we can construct a function g and a sequence $x_n = g(x_{n-1})$ such that we have the sequence $x_n \to x_\infty$ where $g(x_\infty) = x_\infty$. In other words, the sequence of values x_n converges to a fixed point of g. If this fixed point satisfies $f(x_\infty) = 0$, then we have found a solution to our original problem.

The most important algorithm that we will discuss based on functional iteration is Newton's method. The EM algorithm is also an algorithm that can be formulated as a functional iteration and we will discuss that at a later section.

When can such a functional iteration procedure work? The Shrinking Lemma gives us the conditions under which this type of sequence will converge.

2.3.1 The Shrinking Lemma

The Shrinking Lemma gives conditions under which a sequence derived via functional iteration will converge to a fixed point. Let M be a closed subset of a complete normed vector space and let $f : M \to M$ be a map. Assume that there exists a K, $0 < K < 1$, such that for all $x, y \in M$,

$$\|f(x) - f(y)\| \leq K\|x - y\|.$$

Then f has a unique fixed point, i.e. there is a unique point $x_0 \in M$ such that $f(x_0) = x_0$.

Proof: The basic idea of the proof of this lemma is that for a give $x \in M$, we can construct a Cauchy sequence $\{f^n(x)\}$ that converges to x_0, where $f^n(x)$ represents the nth functional iteration of x, i.e. $f^2(x) = f(f(x))$.

Given $x \in M$, we can write

$$\|f^2(x) - f(x)\| = \|f(f(x)) - f(x)\| \leq K\|f(x) - x\|.$$

By induction, we can therefore write

$$\|f(^{n+1}(x) - f^n(x)\| \leq K\|f^n(x) - f^{n-1}(x)\| \leq K^n\|f(x) - x\|.$$

It then follows that

$$\begin{aligned}
\|f^n(x) - x\| &\leq \|f^n(x) - f^{n-1}(x)\| + \|f^{n-1}(x) - f^{n-2}(x)\| + \cdots + \|f(x) - x\| \\
&\leq (K^{n-1} + K^{n-2} + \cdots + K)\|f(x) - x\| \\
&\leq \frac{1}{1-K}\|f(x) - x\|.
\end{aligned}$$

Given integers $m \geq 1$ and $k \geq 1$, we can write

$$\begin{aligned}
\|f^{m+k}(x) - f^k(x)\| &\leq K^m\|f^k(x) - x\| \\
&\leq K^m\frac{1}{1-K}\|f(x) - x\|
\end{aligned}$$

Therefore, there exists some N such that for all $m, n \geq N$ (say $n = m + k$), we have $\|f^n(x) - f^m(x)\| \leq \varepsilon$, because $K^m \to 0$ as $m \to \infty$. As a result, the sequence $\{f^n(x)\}$ is a Cauchy sequence, so let x_0 be its limit.

Given $\varepsilon > 0$, let N be such that for all $n \geq N$, $\|f^n(x) - x_0\| \leq \varepsilon$. Then we can also say that for $n \geq N$,

$$\|f(x_0) - f^{n+1}(x)\| \leq \|x_0 - f^n(x)\| \leq \varepsilon$$

So what we have is $\{f^n(x)\} \to x_0$ and we have $\{f^n(x)\} \to f(x_0)$. Therefore, x_0 is a fixed point of f, so that $f(x_0) = x_0$.

To show that x_0 is unique, suppose that x_1 is another fixed point of f. Then

$$\|x_1 - x_0\| = \|f(x_1) - f(x_0)\| \leq K\|x_1 - x_0\|.$$

Because $0 < K < 1$, we must have $x_0 = x_1$.

2.3.2 Convergence Rates for Shrinking Maps

Suppose g satisfies

$$|g(x) - g(y)| \leq K|x - y|$$

for some $K \in (0, 1)$ and any $x, y \in I$, a closed interval. Therefore, g has a fixesd point at x_∞. Assume g is differentiable with $0 < |g'(x_\infty)| < 1$, where x_∞ is the fixed point of g. Then $x_n \to x_\infty$ linearly.

We can show that

$$\frac{|x_{n+1} - x_\infty|}{|x_n - x_\infty|} = \frac{|g(x_n) - g(x_\infty)|}{|x_n - x_\infty|} \leq K\frac{|x_n - x_\infty|}{|x_n - x_\infty|}.$$

Because $K \in (0, 1)$, this shows that convergence to x_∞ is linear.

2.4 Newton's Method

Newton's method build a sequence of values $\{x_n\}$ via functional iteration that converges to the root of a function f. Let that root be called x_∞ and let x_n be the current estimate. By the mean value theorem, we know there exists some z such that

$$f(x_n) = f'(z)(x_n - x_\infty),$$

where z is somewhere between x_n and x_∞. Rearranging terms, we can write

$$x_\infty = x_n - \frac{f(x_n)}{f'(z)}$$

Obviously, we do not know x_∞ or z, so we can replace them with our next iterate x_{n+1} and our current iterate x_n, giving us the Newton update formula,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

We will discuss Newton's method more in the later section on general optimization, as it is a core method for minimizing functions.

2.4.1 Proof of Newton's Method

Newton's method can be written as a functional iteration that converges to a fixed point. Let f be a function that is twice continuously differentiable and suppose there exists a x_∞ such that $f(x_\infty) = 0$ and $f'(x_\infty) \neq 0$. Then there exists a δ such that for any $x_0 \in (x_\infty - \delta, x_\infty + \delta)$, the sequence

$$x_n = g(x_{n-1}) = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

converges to x_∞.

Note that

$$\begin{aligned} g'(x) &= 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{[f'(x)]^2} \\ &= \frac{f(x)f''(x)}{[f'(x)]^2} \end{aligned}$$

Therefore, $g'(x_\infty) = 0$ because we assume $f(x_\infty) = 0$ and $f'(x_\infty) \neq 0$. Further we know g' is continuous because we assumed f was twice continuously differentiable.

Therefore, given $K < 1$, there exists $\delta > 0$ such that for all $x \in (x_\infty - \delta, x_\infty + \delta)$, we have $|g'(x)| < K$. For any $a, b \in (x_\infty - \delta, x_\infty + \delta)$ we can also write

$$\begin{aligned} |g(a) - g(b)| &\leq |g'(c)||a - b| \\ &\leq K|a - b| \end{aligned}$$

In the interval of $x_\infty \pm \delta$ we have that g is a shrinking map. Therefore, there exists a unique fixed point x_∞ such that $g(x_\infty) = x_\infty$. This value x_∞ is a root of f.

2.4.2 Convergence Rate of Newton's Method

Although proof of Newton's method's convergence to a root can be done using the Shrinking Lemma, the convergence rate of Newton's method is considerably faster than the linear rate of generic shrinking maps. This fast convergence is obtained via the additional assumptions we make about the smoothness of the function f.

Suppose again that f is twice continuously differentiable and that there exists x_∞ such that $f(x_\infty) = 0$. Given some small $\varepsilon > 0$, we can approximate f around x_∞ with

$$\begin{aligned} f(x_\infty + \varepsilon) &= f(x_\infty) + \varepsilon f'(x_\infty) + \frac{\varepsilon^2}{2} f''(x_\infty) + O(\varepsilon^2) \\ &= 0 + \varepsilon f'(x_\infty) + \frac{\varepsilon^2}{2} f''(x_\infty) + O(\varepsilon^2) \end{aligned}$$

Additionally, we can approximate f' with

$$f'(x_\infty + \varepsilon) = f'(x_\infty) + \varepsilon f''(x_\infty) + O(\varepsilon)$$

Recall that Newton's method generates the sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Using the time-honored method of adding and subtracting, we can write this as

$$x_{n+1} - x_\infty = x_n - x_\infty - \frac{f(x_n)}{f'(x_n)}.$$

If we let $\varepsilon_{n+1} = x_{n+1} - x_\infty$ and $\varepsilon_n = x_n - x_\infty$, then we can rewrite the above as

$$\varepsilon_{n+1} = \varepsilon_n - \frac{f(x_n)}{f'(x_n)}$$

Further adding and subtracting (i.e. $x_n = x_\infty + \varepsilon_n$) gives us

$$\varepsilon_{n+1} = \varepsilon_n - \frac{f(x_\infty + \varepsilon_n)}{f'(x_\infty + \varepsilon_n)}$$

From here, we can use the approximations written out earlier to give us

$$\begin{aligned}\varepsilon_{n+1} &\approx \varepsilon_n - \frac{\varepsilon_n f'(x_\infty) + \frac{\varepsilon_n^2}{2} f''(x_\infty)}{f'(x_\infty) + \varepsilon_n f''(x_\infty)} \\ &= \varepsilon_n^2 \left(\frac{\frac{1}{2} f''(x_\infty)}{f'(x_\infty) + \varepsilon_n f''(x_\infty)} \right)\end{aligned}$$

Dividing by ε_n^2 on both sides gives us

$$\frac{\varepsilon_{n+1}}{\varepsilon_n^2} \approx \frac{\frac{1}{2} f''(x_\infty)}{f'(x_\infty) + \varepsilon_n f''(x_\infty)}$$

As $\varepsilon_n \downarrow 0$, we can say that there exists some $M < \infty$ such that

$$\frac{|\varepsilon_{n+1}|}{|\varepsilon_n|^2} \leq M$$

as $n \to \infty$, which is the definition of quadratic convergence. Of course, for this to work we need that $f''(x_\infty) < \infty$ and that $f'(x_\infty) \neq 0$.

In summary, Newton's method is very fast in the neighborhood of the root and furthermore has a direct multivariate generalization (unlike the bisection method). However, the need to evaluate f' at each iteration requires more computation (and more assumptions about the smoothness of f). Additionally, Newton's method can, in a sense, be "too fast" in that there is no guarantee that each iteration of Newton's method is an improvement (i.e. is closer to the root). In certain cases, Newton's method can swing wildly out of control and diverge. Newton's method is only guaranteed to converge in the neighborhood of the root; the exact size of that neighborhood is usually not known.

2.4.3 Newton's Method for Maximum Likelihood Estimation

In many statistical modeling applications, we have a likelihood function L that is induced by a probability distribution that we assume generated the data. This likelihood is typically parameterized by a vector θ and maximizing $L(\theta)$ provides us with the *maximum likelihood estimate* (MLE), or $\hat{\theta}$. In practice, it makes more sense to maximize the log-likehood function, or $\ell(\theta)$, which in many common applications is equivalent to solving the *score equations* $\ell'(\theta) = 0$ for θ.

Newton's method can be applied to generate a sequence that converges to the MLE $\hat{\theta}$. If we assume θ is a $k \times 1$ vector, we can iterate

$$\theta_{n+1} = \theta_n - \ell''(\theta_n)^{-1} \ell'(\theta_n)$$

where ℓ'' is the Hessian of the log-likelihood function.

Note that the formula above computes an inverse of a $k \times k$ matrix, which should serve as an immediate warning sign that this is **not** how the algorithm should be implemented. In practice, it may make more sense to solve the system of equations

$$[\ell''(\theta_n)]\theta_{n+1} = [\ell''(\theta_n)]\theta_n - \ell'(\theta_n).$$

rather than invert $\ell''(\theta_n)$ directly at every iteration.

However, it may make sense to invert $\ell''(\theta_n)$ at the very end of the algorithm to obtain the *observed information matrix* $-\ell''(\hat{\theta})$. This observed information matrix can be used to obtain asymptotic standard errors for $\hat{\theta}$ for making inference about θ.

2.4.3.1 Example: Estimating a Poisson Mean

Suppose we observe data $x_1, x_2, \dots, x_n \overset{iid}{\sim}$ Poisson(μ) and we would like to estimate μ via maximum likelihood. The log-likelihood induced by the Poisson model is

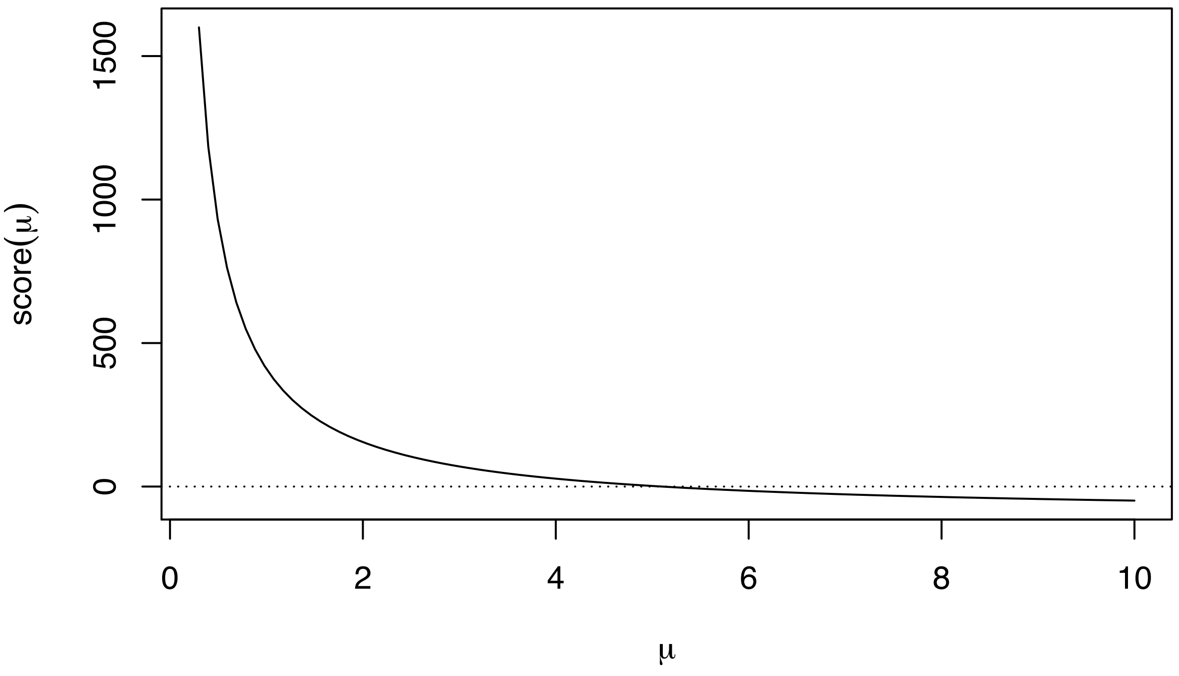
$$\ell(\mu) = \sum_{i=1}^{n} x_i \log \mu - \mu = n\bar{x} \log \mu - n\mu$$

The score function is

$$\ell'(\mu) = \frac{n\bar{x}}{\mu} - n$$

It is clear that setting $\ell'(\mu)$ to zero and solving for μ gives us that $\hat{\mu} = \bar{x}$. However, we can visualizing $\ell'(\mu)$ and see how the Newton iteration would work in this simple scenario.

```
set.seed(2017-08-09)
x <- rpois(100, 5)
xbar <- mean(x)
n <- length(x)
score <- function(mu) {
        n * xbar / mu - n
}
curve(score, .3, 10, xlab = expression(mu), ylab = expression(score(mu)))
abline(h = 0, lty = 3)
```



The figure above shows that this is clearly a nice smooth function for Newton's method to work on. Recall that for the Newton iteration, we also need the second derivative, which in this case is

$$\ell''(\mu) = -\frac{n\bar{x}}{\mu^2}$$

So the Newton iteration is then

$$\begin{aligned}\mu_{n+1} &= \mu_n - \left[-\frac{n\bar{x}}{\mu_n^2}\right]^{-1}\left(\frac{n\bar{x}}{\mu_n} - n\right)\\ &= 2\mu_n - \frac{\mu_n^2}{n}\end{aligned}$$

Using the functional programming aspects of R, we can write a function that executes the functional iteration of Newton's method for however many times we which to run the algorithm.

The following `Iterate()` code takes a function as argument and generates an "iterator" version of it where the number of iterations is an argument.

```
Funcall <- function(f, ...) f(...)
Iterate <- function(f, n = 1) {
        function(x) {
                Reduce(Funcall, rep.int(list(f), n), x, right = TRUE)
        }
}
```

Now we can pass a single iteration of the Newton step as an argument to the `Iterate()` function defined above.

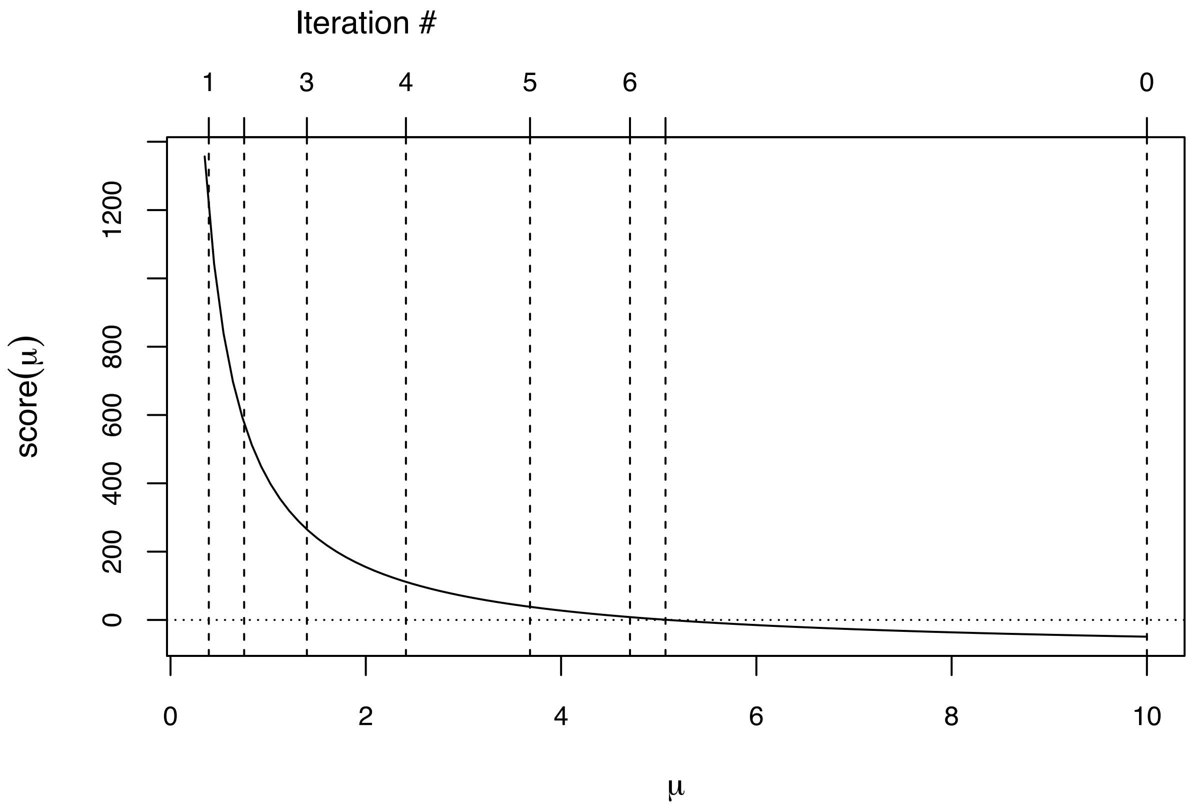
```
single_iteration <- function(x) {
        2 * x - x^2 / xbar
}
g <- function(x0, n) {
        giter <- Iterate(single_iteration, n)
        giter(x0)
}
```

Finally, to facilitate plotting of this function, it is helpful if our iterator function is vectorized with respect to `n`. The `Vectorize()` function can help us here.

```
g <- Vectorize(g, "n")
```

Let's use a starting point of $\mu_0 = 10$. We can plot the score function along with the values of each of the Newton iterates for 7 iterations.

```
par(mar = c(5, 5, 4, 1))
curve(score, .35, 10, xlab = expression(mu), ylab = expression(score(mu)), cex.axis = 0.8)
abline(h = 0, lty = 3)

iterates <- g(10, 1:7)  ## Generate values for 7 functional iterations with a starting value of 10.
abline(v = c(10, iterates), lty = 2)
axis(3, c(10, iterates), labels = c(0, 1:7), cex = 2, cex.axis = 0.8)
mtext("Iteration #", at = 2, line = 2.5)
```



We can see that by the 7th iteration we are quite close to the root, which in this case is 5.1.

Another feature to note of Newton's algorithm here is that when the function is relatively flat, the algorithm makes large moves either to the left or right. However, when the function is relatively steep, the moves are smaller in distance. This makes sense because the size of the deviation from the current iterate depends on the inverse of ℓ'' at the current iterate. When ℓ' is flat, ℓ'' will be small and hence its inverse large.

3 General Optimization

The general optimization problem can be stated as follows. Given a fiunction $f : \mathbb{R}^k \to \mathbb{R}$, we want to find $\min_{x \in S} f(x)$, where $S \subset \mathbb{R}^k$. The general approach to solving this problem that we will discuss is called a *line search method*. With line search methods, given f and a current estimate x_n of the location of the minimum, we want to

1. Choose a direction p_n in k-dimensional space;
2. Choose a step length in the direction p_n, usually by solving $\min_{\alpha>0} f(x_n + \alpha p_n)$ to get α_n
3. Update our estimate with $x_{n+1} = x_n + \alpha_n p_n$.

Clearly then, with line search methods, the two questions one must answer are how should we choose the direction? and how far should we step? Almost all line search approaches provide variations on the answers to those two questions.

Care must be taken in addressing the problems involved with line search methods because typically one must assume that the size of the parameter space is large (i.e. k is large). Therefore, one of constraints for all methods is minimizing the amount of computation that needs to be done due to the large parameter space. Efficiency with respect to memory (storage of data or parameters) and computation time is key.

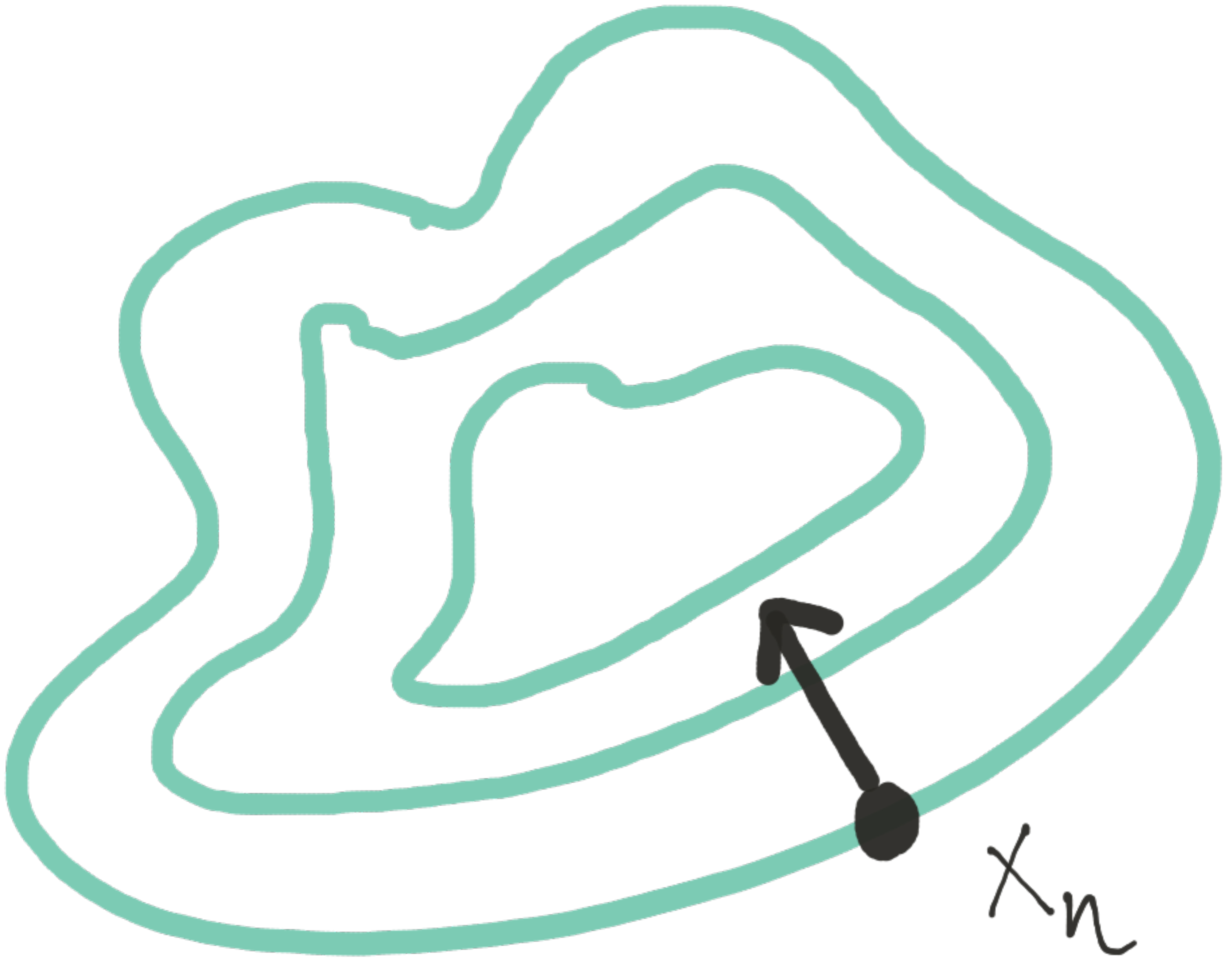


Figure 6: Direction of steepest descent.

3.1 Steepest Descent

Perhaps the most obvious direction to choose when attempting to minimize a function f starting at x_n is the direction of *steepest descent*, or $-f'(x_n)$. This is the direction that is orthogonal to the contours of f at the point x_n and hence is the direction in which f is changing most rapidly at x_n.

The updating procedure for a steepest descent algorithm, given the current estimate x_n, is then

$$x_{n+1} = x_n - \alpha_n f'(x_n)$$

While it might seem logical to always go in the direction of steepest descent, it can occasionally lead to some problems. In particular, when certain parameters are highly correlated with each other, the steepest descent algorithm can require many steps to reach the minimum.

The figure below illustrates a function whose contours are highly correlated and hence elliptical.

Depending on the starting value, the steepest descent algorithm could take many steps to wind its way towards the minimum.

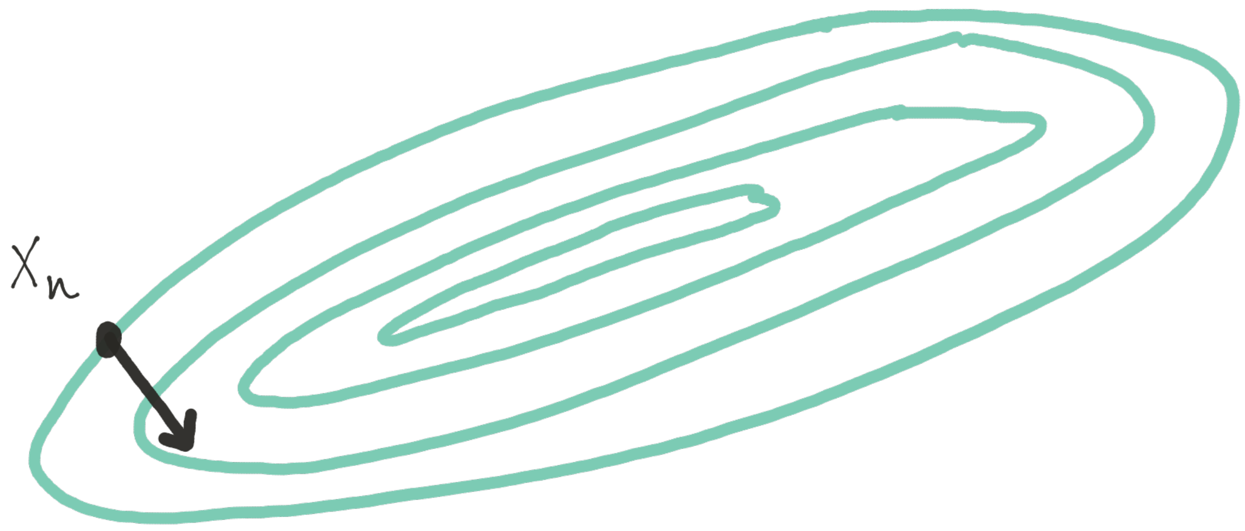


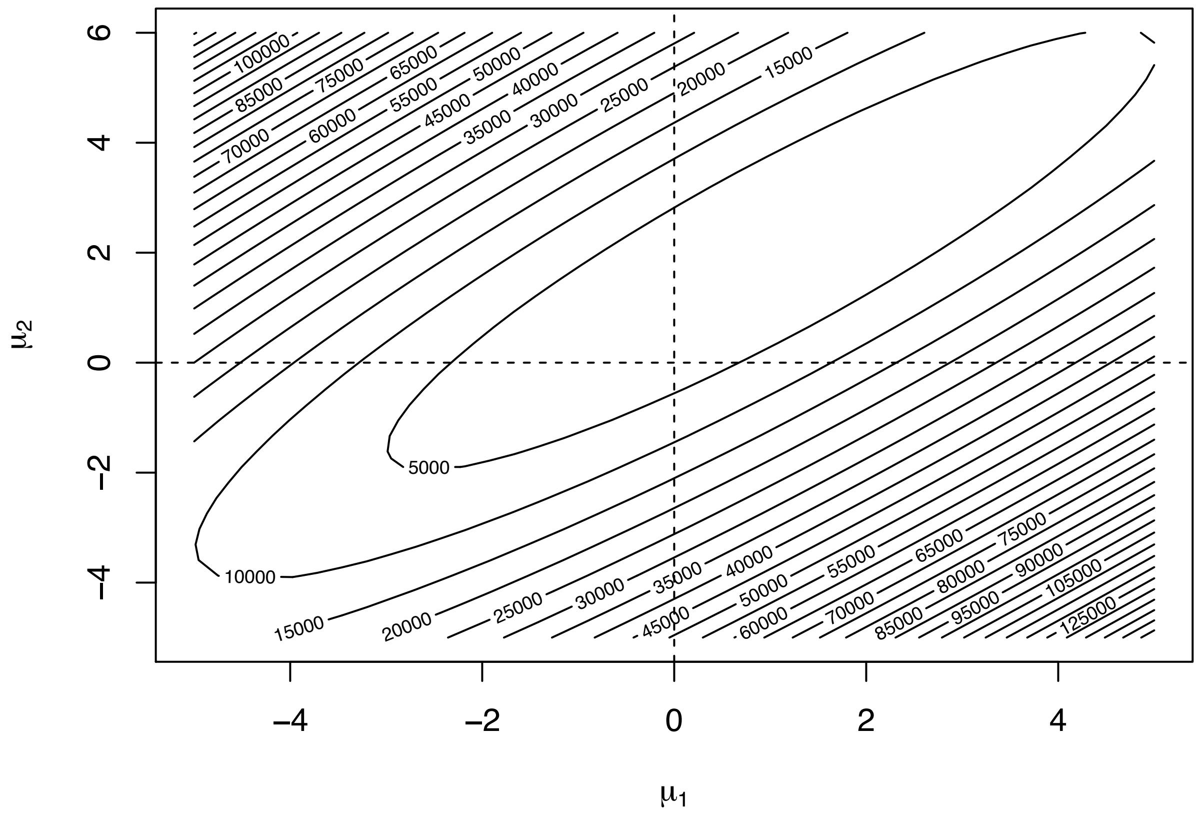
Figure 7: Steepest descent with highly correlated parameters.

3.1.1 Example: Multivariate Normal

One can use steepest descent to compute the maximum likelihood estimate of the mean in a multivariate Normal density, given a sample of data. However, when the data are highly correlated, as they are in the simulated example below, the log-likelihood surface can be come difficult to optimize. In such cases, a very narrow ridge develops in the log-likelihood that can be difficult for the steepest descent algorithm to navigate.

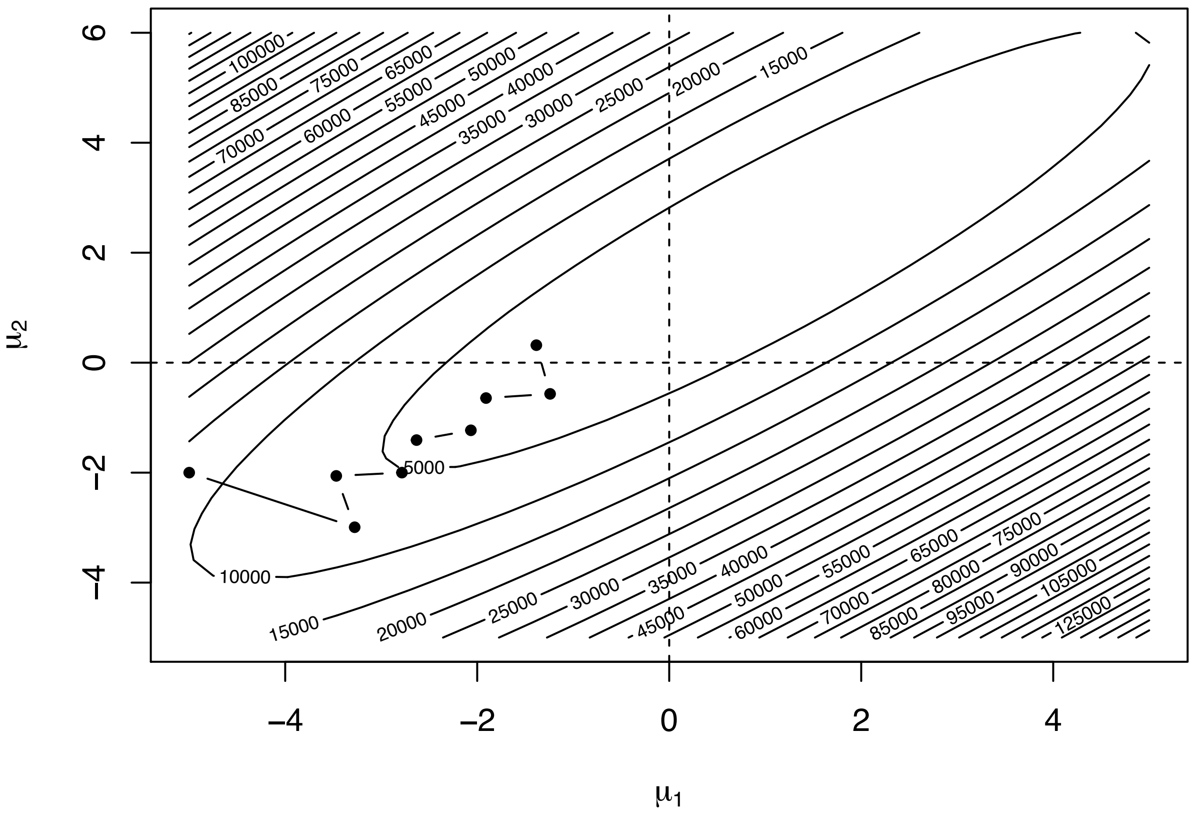
In the example below, we actually compute the *negative* log-likelihood because the algorith is designed to *minimize* functions.

```r
set.seed(2017-08-10)
mu <- c(1, 2)
S <- rbind(c(1, .9), c(.9, 1))
x <- MASS::mvrnorm(500, mu, S)
nloglike <- function(mu1, mu2) {
        dmv <- mvtnorm::dmvnorm(x, c(mu1, mu2), S, log = TRUE)
        -sum(dmv)
}
nloglike <- Vectorize(nloglike, c("mu1", "mu2"))
nx <- 40
ny <- 40
xg <- seq(-5, 5, len = nx)
yg <- seq(-5, 6, len = ny)
g <- expand.grid(xg, yg)
nLL <- nloglike(g[, 1], g[, 2])
z <- matrix(nLL, nx, ny)
par(mar = c(4.5, 4.5, 1, 1))
contour(xg, yg, z, nlevels = 40, xlab = expression(mu[1]),
        ylab = expression(mu[2]))
abline(h = 0, v = 0, lty = 2)
```



Note that in the figure above the surface is highly stretched and that the minimum $(1, 2)$ lies in the middle of a narrow valley. For the steepest descent algorithm we will start at the point $(-5, -2)$ and track the path of the algorithm.

```
library(dplyr, warn.conflicts = FALSE)
norm <- function(x) x / sqrt(sum(x^2))
Sinv <- solve(S)  ## I know I said not to do this!
step1 <- function(mu, alpha = 1) {
        D <- sweep(x, 2, mu, "-")
        score <- colSums(D) %>% norm
        mu + alpha * drop(Sinv %*% score)
}
steep <- function(mu, n = 10, ...) {
        results <- vector("list", length = n)
        for(i in seq_len(n)) {
                results[[i]] <- step1(mu, ...)
                mu <- results[[i]]
        }
        results
}
m <- do.call("rbind", steep(c(-5, -2), 8))
m <- rbind(c(-5, -2), m)

par(mar = c(4.5, 4.5, 1, 1))
contour(xg, yg, z, nlevels = 40, xlab = expression(mu[1]),
        ylab = expression(mu[2]))
abline(h = 0, v = 0, lty = 2)
points(m, pch = 20, type = "b")
```



We can see that the path of the algorthm is rather winding as it traverses the narrow valley. Now, we have fixed the step-length in this case, which is probably not optimal. However, one can still see that the algorithm has some difficulty navigating the surface because the direction of steepest descent does not take one directly towards the minimum ever.

3.2 The Newton Direction

Given a current best estimate x_n, we can approximate f with a quadratic polynomial. For some small p,

$$f(x_n + p) \approx f(x_n) + p' f'(x_n) + \frac{1}{2} p' f''(x_n) p.$$

If we minimize the right hand side with respect to p, we obtain

$$p_n = f''(x_n)^{-1}[-f'(x_n)]$$

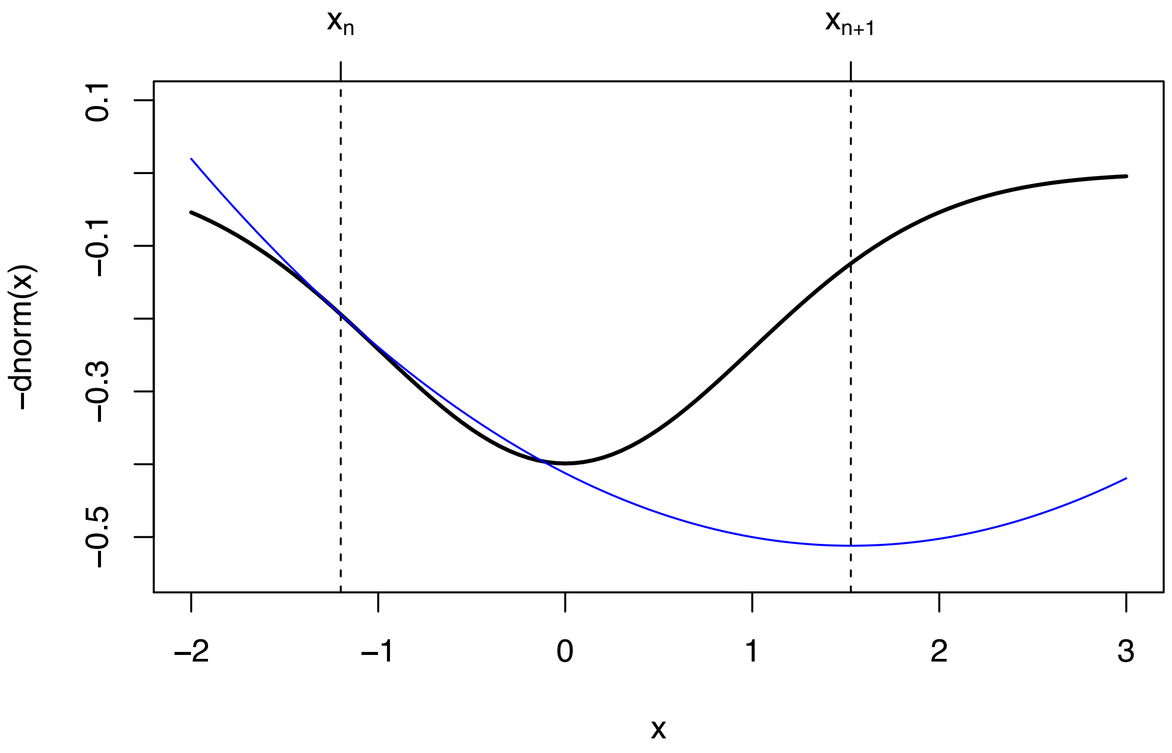
which we can think of as the steepest descent direction "twisted" by the inverse of the Hessian matrix $f''(x_n)^{-1}$. Newton's method has a "natural" step length of 1, so that the updating procedure is

$$x_{n+1} = x_n - f''(x_n)^{-1} f'(x_n).$$

Newton's method makes a quadratic approximation to the target function f at each step of the algorithm. This follows the "optimization transfer" principle mentioned earlier, whereby we take a complex function f, replace it with a simpler function g that is easier to optimize, and then optimize the simpler function repeatedly until convergence to the solution.

We can visualize how Newton's method makes its quadratic approximation to the target function easily in one dimension.

```
curve(-dnorm(x), -2, 3, lwd = 2, ylim = c(-0.55, .1))
xn <- -1.2
abline(v = xn, lty = 2)
axis(3, xn, expression(x[n]))
g <- function(x) {
        -dnorm(xn) + (x-xn) * xn * dnorm(xn) - 0.5 * (x-xn)^2 * (dnorm(xn) - xn * (xn * dnorm(xn)))
}
curve(g, -2, 3, add = TRUE, col = 4)
op <- optimize(g, c(0, 3))
abline(v = op$minimum, lty = 2)
axis(3, op$minimum, expression(x[n+1]))
```



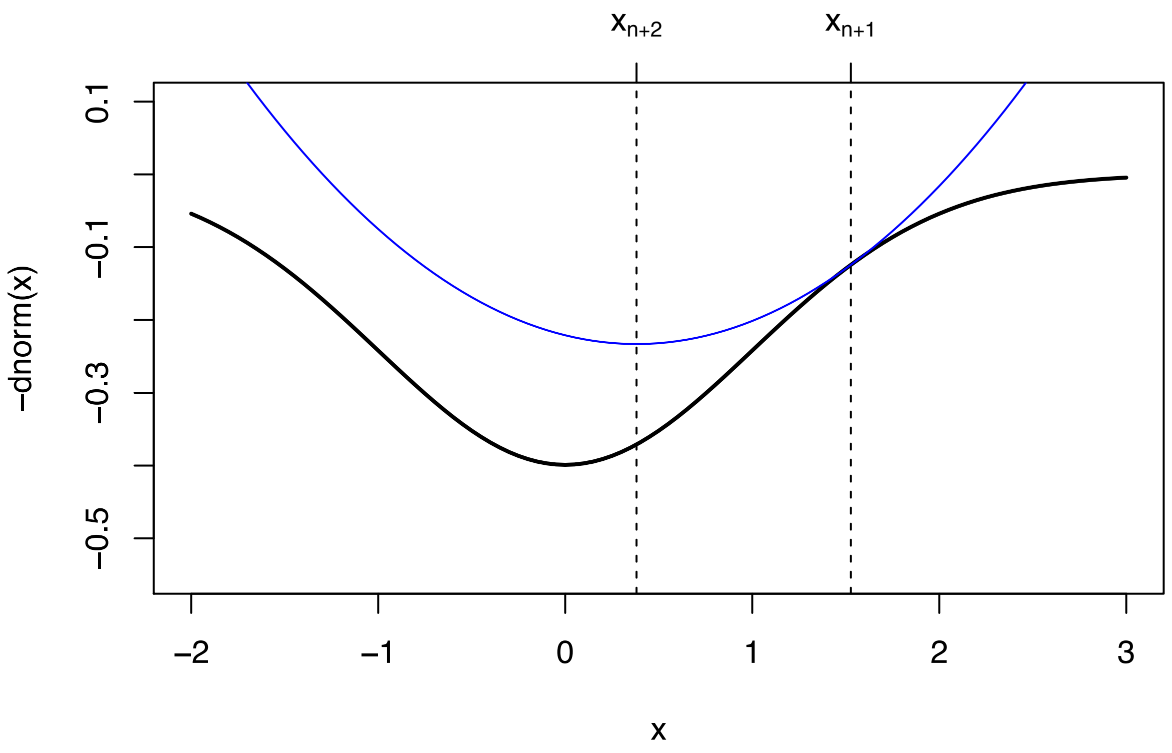
In the above figure, the next iterate, x_{n+1} is actually further away from the minimum than our previous iterate x_n. The quadratic approximation that Newton's method makes to f is not guaranteed to be good at every point of the function.

This shows an important "feature" of Newton's method, which is that it is not *monotone*. The successive iterations that Newton's method produces are not guaranteed to be improvements in the sense that each iterate is closer to the truth. The tradeoff here is that while Newton's method is very fast (quadratic convergence), it can be unstable at times. Monotone algorithms (like the EM algorithm that we discuss later) that always produce improvements, are more stable, but generally converge at slower rates.

In the next figure, however, we can see that the solution provided by the next approximation, x_{n+2}, is indeed quite close to the true minimum.

```
curve(-dnorm(x), -2, 3, lwd = 2, ylim = c(-0.55, .1))
xn <- -1.2
op <- optimize(g, c(0, 3))
abline(v = op$minimum, lty = 2)
axis(3, op$minimum, expression(x[n+1]))

xn <- op$minimum
```

```r
curve(g, -2, 3, add = TRUE, col = 4)
op <- optimize(g, c(0, 3))
abline(v = op$minimum, lty = 2)
axis(3, op$minimum, expression(x[n+2]))
```



It is worth noting that in the rare event that f is in fact a quadratic polynomial, Newton's method will converge in a single step because the quadratic approximation that it makes to f will be exact.

3.2.1 Generalized Linear Models

The generalized linear model is an extension of the standard linear model to allow for non-Normal response distributions. The distributions used typically come from an exponential family whose density functions share some common characteristics. With a GLM, we typical present it as $y_i \sim p(y_i \mid \mu_i)$, where p is an exponential family distribution, $\mathbb{E}[y_i] = \mu_i$,

$$g(\mu_i) = x_i'\beta,$$

where g is a nonlinear link function, and $\text{Var}(y_i) = V(\mu)$ where V is a known variance function.

Unlike the standard linear model, the maximum likelihood estimate of the parameter vector β cannot be obtained in closed form, so an iterative algorithm must be used to obtain the estimate. The traditional algorithm used is the Fisher scoring algorithm. This algorithm uses a linear approximation to the nonlinear link function g, which can be written as

$$g(y_i) \approx g(\mu_i) + (y_i - \mu_i)g'(\mu_i).$$

The typical notation of GLMs refers to $z_i = g(\mu_i) + (y_i - \mu_i)g'(\mu_i)$ as the *working response*. The Fisher scoring algorithm then works as follows.

1. Start with $\hat{\mu}_i$, some initial value.
2. Compute $z_i = g(\hat{\mu}_i) + (y_i - \hat{\mu}_i)g'(\hat{\mu}_i)$.

3. Given the $n \times 1$ vector of working responses z and the $n \times p$ predictor matrix X we compute a weighted regression of z on X to get

$$\beta_n = (X'WX)^{-1}X'Wz$$

where W is a diagonal matrix with diagonal elements

$$w_{ii} = \left[g'(\mu_i)^2 V(\mu_i)\right]^{-1}.$$

4. Given β_n, we can recompute $\hat{\mu}_i = g^{-1}(x_i'\beta_n)$ and go to 2.

Note that in Step 3 above, the weights are simply the inverses of the variance of z_i, i.e.

$$\begin{aligned} \text{Var}(z_i) &= \text{Var}(g(\mu_i) + (y_i - \mu_i)g'(\mu_i)) \\ &= \text{Var}((y_i - \mu_i)g'(\mu_i)) \\ &= V(\mu_i)g'(\mu_i)^2 \end{aligned}$$

Naturally, when doing a weighted regression, we would weight by the inverse of the variances.

3.2.1.1 Example: Poisson Regression

For a Poisson regression, we have $y_i \sim \text{Poisson}(\mu_i)$ where $g(\mu) = \log \mu_i = x_i'\beta$ because the log is the canonical link function for the Poisson distribution. We also have $g'(\mu_i) = \frac{1}{\mu_i}$ and $V(\mu_i) = \mu_i$. Therefore, the Fisher scoring algorithm is

1. Initialize $\hat{\mu}_i$, perhaps using $y_i + 1$ (to avoid zeros).
2. Let $z_i = \log \hat{\mu}_i + (y_i - \hat{\mu}_i)\frac{1}{\hat{\mu}_i}$
3. Regression z on X using the weights

$$w_{ii} = \left[\frac{1}{\hat{\mu}_i^2}\hat{\mu}_i\right]^{-1} = \hat{\mu}_i.$$

Using the Poisson regression example, we can draw a connection between the usual Fisher scoring algorithm for fitting GLMs and Newton's method. Recall that if $\ell(\beta)$ is the log-likelihood as a function of the regression paramters β, then the Newton updating scheme is

$$\beta_{n+1} = \beta_n + \ell''(\beta_n)^{-1}[-\ell'(\beta_n)].$$

The log-likelihoood for a Poisson regression model can be written in vector/matrix form as

$$\ell(\beta) = y'X\beta - \exp(X\beta)'\mathbf{1}$$

where the exponential is taken component-wise on the vector $X\beta$. The gradient function is

$$\ell'(\beta) = X'y - X'\exp(X\beta) = X'(y - \mu)$$

and the Hessian is

$$\ell''(\beta) = -X'WX$$

where W is a diagonal matrix with the values $w_{ii} = \exp(x_i'\beta)$ on the diagonal. The Newton iteration is then

$$\begin{aligned} \beta_{n+1} &= \beta_n + (-X'WX)^{-1}(-X'(y - \mu)) \\ &= \beta_n + (X'WX)^{-1}XW(z - X\beta_n) \\ &= (X'WX)^{-1}X'Wz + \beta_n - (X'WX)^{-1}X'WX\beta_n \\ &= (X'WX)^{-1}X'Wz \end{aligned}$$

Therefore the iteration is exactly the same as the Fisher scoring algorithm in this case. In general, Newton's method and Fisher scoring will coincide with any generalized linear model using an exponential family with a canonical link function.

3.2.2 Newton's Method in R

The `nlm()` function in R implements Newton's method for minimizing a function given a vector of starting values. By default, one does not need to supply the gradient or Hessian functions; they will be estimated numerically by the algorithm. However, for the purposes of improving accuracy of the algorithm, both the gradient and Hessian can be supplied as attributes of the target function.

As an example, we will use the `nlm()` function to fit a simple logistic regression model for binary data. This model specifies that $y_i \sim \text{Bernoulli}(p_i)$ where

$$\log \frac{p_i}{1-p_i} = \beta_0 + x_i\beta_1$$

and the goal is to estimate β via maximum likelihood. Given the assumed Bernoulli distribution, we can write the log-likelihood for a single observation as

$$\begin{aligned}
\log L(\beta) &= \log\left\{\prod_{i=1}^n p_i^{y_i}(1-p_i)^{1-y_i}\right\} \\
&= \sum_{i=1}^n y_i \log p_i + (1-y_i)\log(1-p_i) \\
&= \sum_{i=1}^n y_i \log \frac{p_i}{1-p_i} + \log(1-p_i) \\
&= \sum_{i=1}^n y_i(\beta_0 + x_i\beta_1) + \log\left(\frac{1}{1+e^{(\beta_0+x_i\beta_1)}}\right) \\
&= \sum_{i=1}^n y_i(\beta_0 + x_i\beta_1) - \log\left(1+e^{(\beta_0+x_i\beta_1)}\right)
\end{aligned}$$

If we take the very last line of the above derivation and take a single element inside the sum, we have

$$\ell_i(\beta) = y_i(\beta_0 + x_i\beta_1) - \log\left(1+e^{(\beta_0+x_i\beta_1)}\right)$$

We will need the gradient and Hessian of this with respect to β. Because the sum and the derivative are exchangeable, we can then sum each of the individual gradients and Hessians to get the full gradient and Hessian for the entire sample, so that

$$\ell'(\beta) = \sum_{i=1}^n \ell_i'(\beta)$$

and

$$\ell''(\beta) = \sum_{i=1}^n \ell_i''(\beta).$$

Now, taking the gradient and Hessian of the above expression may be mildly inconvenient, but it is far from impossible. Nevertheless, R provides an automated way to do symbolic differentiation so that manual work can be avoided. The `deriv()` function computes the gradient and Hessian of an expression symbolically so that it can be used in minimization routines. It cannot compute gradients of arbitrary expressions, but it it does support a wide range of common statistical functions.

3.2.2.1 Example: Trends in p-values Over Time

The `tidypvals` package written by Jeff Leek contains datasets taken from the literature collecting p-values associated with various publications along with some information about those publications (i.e. journal, year, DOI). One question that comes up is whether there has been any trend over time in the claimed statistical significance of publications, where "statistical significance" is defined as having a p-value less than 0.05.

The `tidypvals` package is available from GitHub and can be installed using the `install_github()` function in the `remotes` package.

```
remotes::install_github("jtleek/tidypvals")
```

Once installed, we will make use of the `jager2014` dataset. In particular, we are interseted in creating an indicator of whether a p-value is less than 0.05 and regressing it on the `year` variable.

```
library(tidypvals)
library(dplyr)
jager <- mutate(tidypvals::jager2014,
                pvalue = as.numeric(as.character(pvalue)),
                y = ifelse(pvalue < 0.05
                           | (pvalue == 0.05 & operator == "lessthan"),
                           1, 0),
                x = year - 2000) %>%
        tbl_df
```

Note here that we have subtracted the year 2000 off of the `year` variable so that $x = 0$ corresponds to `year == 2000`.

Next we compute the gradient and Hessian of the negative log-likelihood with respect to β_0 and β_1 using the `deriv()` function. Below, we specify `function.arg = TRUE` in the call to `deriv()` because we want `deriv()` to return a *function* whose arguments are `b0` and `b1`.

```
nll_one <- deriv(~ -(y * (b0 + x * b1) - log(1 + exp(b0 + b1 * x))),
             c("b0", "b1"), function.arg = TRUE, hessian = TRUE)
```

Here's what that function looks like.

```
nll_one
```

```
function (b0, b1)
{
    .expr6 <- exp(b0 + b1 * x)
    .expr7 <- 1 + .expr6
    .expr11 <- .expr6/.expr7
    .expr15 <- .expr7^2
    .expr18 <- .expr6 * x
    .expr19 <- .expr18/.expr7
    .value <- -(y * (b0 + x * b1) - log(.expr7))
    .grad <- array(0, c(length(.value), 2L), list(NULL, c("b0",
        "b1")))
    .hessian <- array(0, c(length(.value), 2L, 2L), list(NULL,
        c("b0", "b1"), c("b0", "b1")))
    .grad[, "b0"] <- -(y - .expr11)
    .hessian[, "b0", "b0"] <- .expr11 - .expr6 * .expr6/.expr15
    .hessian[, "b0", "b1"] <- .hessian[, "b1", "b0"] <- .expr19 -
        .expr6 * .expr18/.expr15
    .grad[, "b1"] <- -(y * x - .expr19)
    .hessian[, "b1", "b1"] <- .expr18 * x/.expr7 - .expr18 *
```

```
        .expr18/.expr15
    attr(.value, "gradient") <- .grad
    attr(.value, "hessian") <- .hessian
    .value
}
```

The function `nll_one()` produced by `deriv()` evaluates the negative log-likelihood for each data point. The output from `nll_one()` will have attributes `"gradient"` and `"hessian"` which represent the gradient and Hessian, respectively. For example, using the data from the `jager` dataset, we can evaluate the negative log-likelihood at $\beta_0 = 0, \beta_1 = 0$.

```
x <- jager$x
y <- jager$y
str(nll_one(0, 0))
```

```
 num [1:15653] 0.693 0.693 0.693 0.693 0.693 ...
 - attr(*, "gradient")= num [1:15653, 1:2] -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:2] "b0" "b1"
 - attr(*, "hessian")= num [1:15653, 1:2, 1:2] 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 ...
  ..- attr(*, "dimnames")=List of 3
  .. ..$ : NULL
  .. ..$ : chr [1:2] "b0" "b1"
  .. ..$ : chr [1:2] "b0" "b1"
```

The `nll_one()` function evaluates the negative log-likelihood at each data point, but does not sum the points up as would be required to evaluate the full negative log-likelihood. Therefore, we will write a separate function that does that for the negative log-likelihood, gradient, and Hessian.

```
nll <- function(b) {
        v <- nll_one(b[1], b[2])
        f <- sum(v)                                     ## log-likelihood
        gr <- colSums(attr(v, "gradient"))              ## gradient vector
        hess <- apply(attr(v, "hessian"), c(2, 3), sum) ## Hessian matrix
        attributes(f) <- list(gradient = gr,
                              hessian = hess)
        f
}
```

Now, we can evaluate the full negative log-likelihood with the `nll()` function. Note that `nll()` takes a single numeric vector as input as this is what the `nlm()` function is expecting.

```
nll(c(0, 0))
```

```
[1] 10849.83
attr(,"gradient")
      b0       b1
 -4586.5 -21854.5
attr(,"hessian")
         b0        b1
b0  3913.25  19618.25
b1 19618.25 137733.75
```

Using $\beta_0 = 0, \beta_1 = 0$ as the initial value, we can call `nlm()` to minimize the negative log-likelihood.

```
res <- nlm(nll, c(0, 0))
res
```

```
$minimum
[1] 7956.976

$estimate
[1]  1.57032807 -0.04416515

$gradient
[1] -0.000001451746 -0.000002782241

$code
[1] 1

$iterations
[1] 4
```

Note first in the output that there is a `code` with the value `4` and that the number of iterations is 100. Whenever the number of iterations in an optimization algorithm is a nice round number, the chances are good that it it some preset iteration limit. This in turn usually means the algorithm didn't converge.

In the help for `nlm()` we also learn that the `code` value of `4` means "iteration limit exceeded", which is generally not good. Luckily, the solution is simple: we can increase the iteration limit and let the algorithm run longer.

```r
res <- nlm(nll, c(0, 0), iterlim = 1000)
res
```

```
$minimum
[1] 7956.976

$estimate
[1]  1.57032807 -0.04416515

$gradient
[1] -0.000001451746 -0.000002782241

$code
[1] 1

$iterations
[1] 4
```

Here we see that the number of iterations used was 260, which is well below the iteration limit. Now we get `code` equal to 2 which means that "successive iterates within tolerance, current iterate is probably solution". Sounds like good news!

Lastly, most optimization algorithms have an option to scale your parameter values so that they roughly vary on the same scale. If your target function has paramters that vary on wildly different scales, this can cause a practical problem for the computer (it's not a problem for the theory). The way to deal with this in `nlm()` is to use the `typsize` arguemnt, which is a vector equal in length to the parameter vector which provides the relative sizes of the parameters.

Here, I give `typsize = c(1, 0.1)`, which indicates to `nlm()` that the first paramter, β_0, should be roughly 10 times larger than the second parameter, β_1 when the target function is at its minimum.

```r
res <- nlm(nll, c(0, 0), iterlim = 1000,
           typsize = c(1, 0.1))
res
```

```
$minimum
[1] 7956.976

$estimate
[1]  1.57032807 -0.04416515

$gradient
[1] -0.000001451745 -0.000002782238

$code
[1] 1

$iterations
[1] 4
```

Running this call to `nlm()` you'll notice that the solution is the same but the number of iterations is actually much less than before (4 iterations) which means the algorithm ran faster. Generally speaking, scaling the parameter vector appropriately (if possible) improves the performance of all optimization algorithms and in my experience is almost always a good idea. The specific values given to the `typsize` argument are not important; rather their relationships to each other (i.e. orders of magnitude) are what matter.

3.3 Quasi-Newton

Quasi-Newton methods arise from the desire to use something like Newton's method for its speed but without having to compute the Hessian matrix each time. The idea is that if the Newton iteration is

$$\theta_{n+1} = \theta_n - f''(\theta_n)^{-1} f'(\theta_n)$$

is there some other matrix that we can use to replace either $f''(\theta_n)$ or $f''(\theta_n)^{-1}$? That is can we use a revised iteration,

$$\theta_{n+1} = \theta_n - B_n^{-1} f'(\theta_n)$$

where B_n is simpler to compute but still allows the algorithm to converge quickly?

This is a challenging problem because $f''(\theta_n)$ gives us a lot of information about the surface of f at θ_n and throwing out this information results in, well, a severe loss of information.

The idea with Quasi-Newton is to find a solution B_n to the problem

$$f'(\theta_n) - f'(\theta_{n-1}) = B_n(\theta_n - \theta_{n-1}).$$

The equation above is sometimes referred to as the *secant equation.* Note first that this requires us to store two values, θ_n and θ_{n-1}. Also, in one dimension, the solution is trivial: we can simply divide the left-hand-side by $\theta_n - \theta_{n-1}$. However, in more than one dimension, there exists an infinite number of solutions and we need some way to constrain the problem to arrive at a sensible answer.

The key to Quasi-Newton approaches in general is that while we initially may not have much information about f, with each iteration we obtain just a little bit more. Specifically, we learn more about the Hessian matrix through successive differences in f'. Therefore, with each iteration we can incorporate this newly obtained information into our estimate of the Hessian matrix. The constraints placed on the matrix B_n is that it be *symmetric* and that it be close to B_{n-1}. These constraints can be satisfied by updating B_n via the addition of rank one matrices.

If we let $y_n = f'(\theta_n) - f'(\theta_{n-1})$ and $s_n = \theta_n - \theta_{n-1}$, then the secant equation is $y_n = B_n s_n$. One updating procedures for B_n

$$B_n = B_{n-1} + \frac{y_n y_n'}{y_n' s_n} - \frac{B_{n-1} s_n s_n' B_{n-1}'}{s_n' B_{n-1} s_n}$$

The above updating procedure was developed by Broyden, Fletcher, Goldfarb, and Shanno (BFGS). An analogous approach, which solves the following secant equation, $H_n y_n = s_n$ was proposed by Davidon, Fletcher, and Powell (DFP).

Note that in the case of the BFGS method, we actually use B_n^{-1} in the Newton update. However, it is not necessary to solve for B_n and then invert it directly. We can directly update B_{n-1}^{-1} to produce B_n^{-1} via the Sherman-Morrison update formula. This formula allows us to generate the new inverse matrix by using the previous inverse and some matrix multiplication.

3.3.1 Quasi-Newton Methods in R

Quasi-Newton methods in R can be accessed through the `optim()` function, which is a general purpose optimization function. The `optim()` function implements a variety of methods but in this section we will focus on the `"BFGS"` and `"L-BFGS-B"`methods.

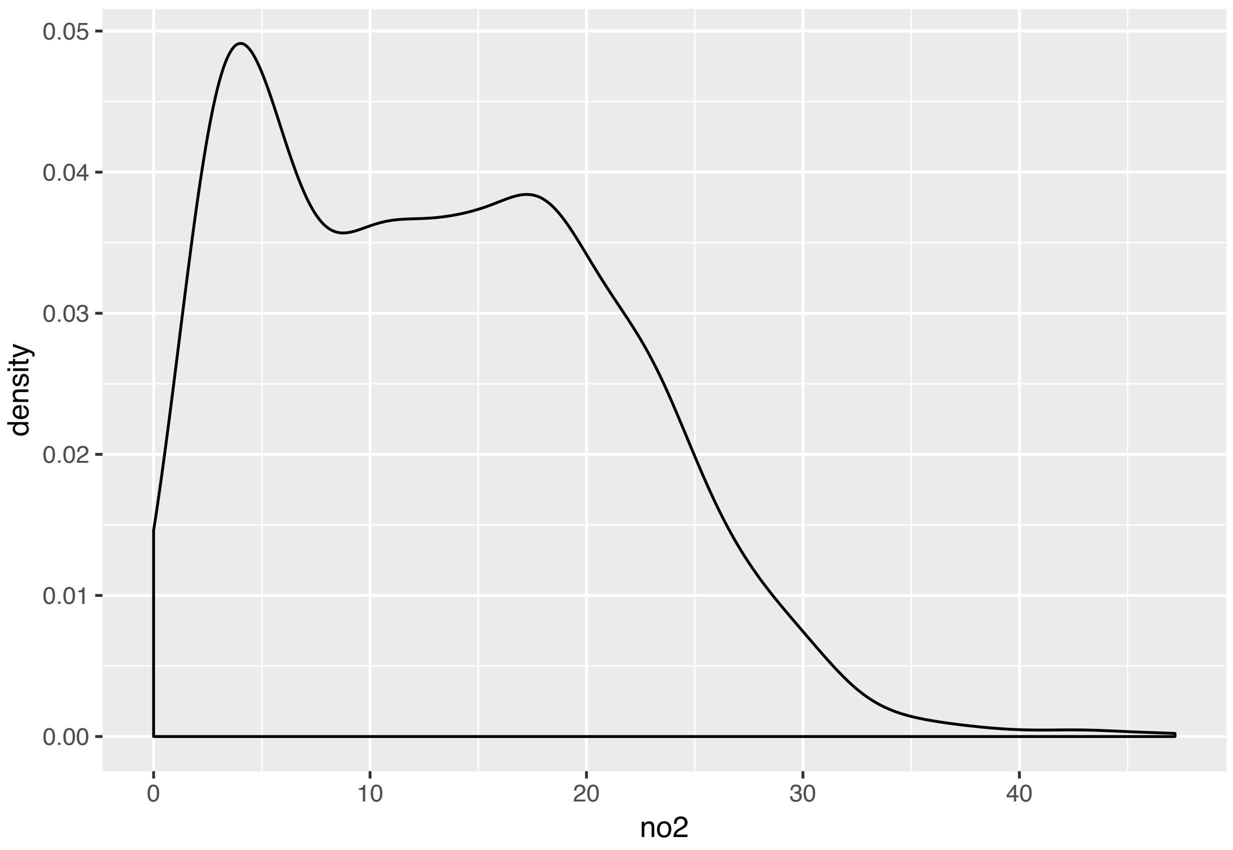
3.3.1.1 Example: Truncated Normal and Mixture of Normal Distributions

The data were obtained from the U.S. Environmental Protection Agency's Air Quality System web page. For this example we will be using the daily average concentrations of nitrogen dioxide (NO_2) for 2016 found in this file. In particular, we will focus on the data for monitors located in Washington State.

```r
library(readr)
library(tidyr)
dat0 <- read_csv("daily_42602_2016.csv.bz2")
names(dat0) <- make.names(names(dat0))
dat <- filter(dat0, State.Name == "Washington") %>%
        unite(site, State.Code, County.Code, Site.Num) %>%
        rename(no2 = Arithmetic.Mean, date = Date.Local) %>%
        select(site, date, no2)
```

A kernel density estimate of the NO_2 data shows the following distribution.

```r
library(ggplot2)
ggplot(dat, aes(x = no2)) +
        geom_density()
```



As an initial stab at characterizing the distribution of the NO$_2$ values (and to demonstrate the use of `optim()` for fitting models), we will try to fit a truncated Normal model to the data. The truncated Normal can make sense for these kinds of data because they are strictly positive, making a standard Normal distribution inappropriate.

For the truncated normal, truncated from below at 0, the density of the data is

$$f(x) = \frac{\frac{1}{\sigma}\varphi\left(\frac{x-\mu}{\sigma}\right)}{\int_0^\infty \frac{1}{\sigma}\varphi\left(\frac{x-\mu}{\sigma}\right)\,dx}.$$

The unknown parameters are μ and σ. Given the density, we can attempt to estimate μ and σ by maximum likelihood. In this case, we will minimize the *negative* log-likelihood of the data.

We can use the `deriv()` function to compute the negative log-likelihood and its gradient automatically. Because we are using quasi-Newton methods here we do not need the Hessian matrix.

```
nll_one <- deriv(~ -log(dnorm((x - mu)/s) / s) + log(0.5),
                 c("mu", "s"),
                 function.arg = TRUE)
```

The `optim()` function works a bit differently from `nlm()` in that instead of having the gradient as an attribute of the negative log-likelhood, the gradient needs to be a separate function.

First the negative log-likelihood.

```
nll <- function(p) {
        v <- nll_one(p[1], p[2])
        sum(v)
}
```

Then the gradient function.

```
nll_grad <- function(p) {
        v <- nll_one(p[1], p[2])
        colSums(attr(v, "gradient"))
}
```

Now we can pass the `nll()` and `nll_grad()` functions to `optim()` to obtain estimates of μ and σ. We will use starting values of $\mu = 1$ and $\sigma = 5$. To use the `"BFGS"` quasi-Newton method you need to specify it in the `method` argument. The default method for `optim()` is the Nelder-Mead simplex method. We also specify `hessian = TRUE` to tell `optim()` to numerically calculate the Hessian matrix at the optimum point.

```
x <- dat$no2
res <- optim(c(1, 5), nll, gr = nll_grad,
             method = "BFGS", hessian = TRUE)
```

```
Warning in log(.expr4): NaNs produced

Warning in log(.expr4): NaNs produced

Warning in log(.expr4): NaNs produced
```

```
res
```

```
$par
[1] 13.23731  8.26315

$value
[1] 4043.641

$counts
function gradient
      35       19

$convergence
[1] 0

$message
NULL

$hessian
              [,1]         [,2]
[1,] 20.8700535980  0.0005659674
[2,]  0.0005659674 41.7458205694
```

The `optim()` function returns a list with 5 elements (plus a Hessian matrix if `hessian = TRUE` is set). The first element that you should check is the `onvergence` code. If `convergece` is 0, that is good. Anything other than 0 could indicate a problem, the nature of which depends on the algorithm you are using (see the help page for `optim()` for more details). This time we also had `optim()` compute the Hessian (numerically) at the optimal point so that we could derive asymptotic standard errors if we wanted.

First note that there were a few messages printed to the console while the algorithm was running indicating that `NaN`s were produced by the target function. This is likely because the function was attempting to take the log of negative numbers. Because we used the `"BFGS"` algorithm, we were conducting an unconstrained optimization. Therefore, it's possible that the algorithm's search produced negative values for σ, which don't make sense in this context. In order to constrain the search, we can use the `"L-BFGS-B"` methods which is a "limited memory" BFGS algorithm with "box constraints". This allows you to put a lower and upper bound on each parameter in the model.

Note that `optim()` allows your target function to produce `NA` or `NaN` values, and indeed from the output it seems that the algorithm eventually converged on the answer anyway. But since we know that the parameters in this model are constrained, we can go ahead and use the alternate approach.

Here we set the lower bound for all parameters to be 0 but allow the upper bound to be infinity (`Inf`), which is the default.

```
res <- optim(c(1, 5), nll, gr = nll_grad,
             method = "L-BFGS-B", hessian = TRUE,
             lower = 0)
res
```

```
$par
[1] 13.237470  8.263546

$value
[1] 4043.641

$counts
function gradient
      14       14

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"

$hessian
             [,1]         [,2]
[1,] 20.868057205 -0.000250771
[2,] -0.000250771 41.735838073
```

We can see now that the warning messages are gone, but the solution is identical to that produced by the original `"BFGS"` method.

The maximum likelihood estimate of μ is 13.24 and the estimate of σ is 8.26. If we wanted to obtain asymptotic standard errors for these parameters, we could look at the Hessian matrix.

```
solve(res$hessian) %>%
        diag %>%
        sqrt
```

```
[1] 0.2189067 0.1547909
```

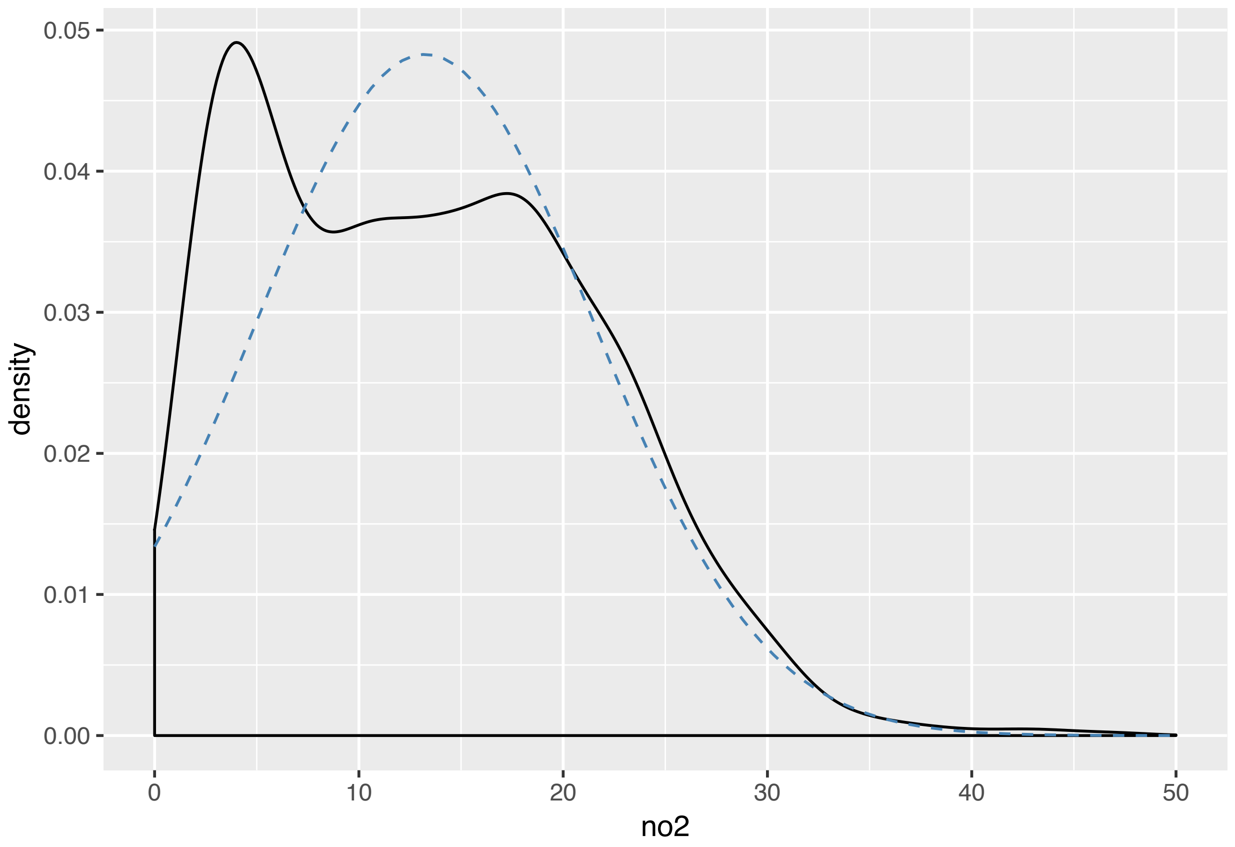
In this case though, we don't care much for the standard errors so we will move on.

We can plot the original density smooth of the data versus the fitted truncated Normal model to see how well we charaterize the distribution. First we will evaluate the fitted model at 100 points between 0 and 50.

```
xpts <- seq(0, 50, len = 100)
dens <- data.frame(xpts = xpts,
                   ypts = dnorm(xpts, res$par[1], res$par[2]))
```

Then we can overlay the fitted model on top of the density using `geom_line()`.

```
ggplot(dat, aes(x = no2)) +
        geom_density() +
        geom_line(aes(x = xpts, y = ypts), data = dens, col = "steelblue",
                    lty = 2)
```



It's not a great fit. Looking at the density smooth of the data, it's clear that there are two modes to the data, suggesting that a truncated Normal might not be sufficient to characterize the data.

One alternative in this case would be a mixture of two Normals, which might capture the two modes. For a two-component mixture, the density for the data would be

$$f(x) = \lambda \frac{1}{\sigma}\varphi\left(\frac{x-\mu_1}{\sigma_1}\right) + (1-\lambda)\frac{1}{\sigma}\varphi\left(\frac{x-\mu_2}{\sigma_2}\right).$$

Commonly, we see that this model is fit using more complex algorithms like the EM algorithm or Markov chain Monte Carlo methods. While those methods do provide greater stability in the estimation process (as we will see later), we can in fact use Newton-type methods to maximize the likelihood directly with a little care.

First we can write out the negative log-likelihood symbolically and allow R's `deriv()` function to compute the gradient function.

```
nll_one <- deriv(~ -log(lambda * dnorm((x-mu1)/s1)/s1 + (1-lambda)*dnorm((x-mu2)/s2)/s2),
                 c("mu1", "mu2", "s1", "s2", "lambda"),
                 function.arg = TRUE)
```

Then, as before, we can specify separate negative log-likelihood (`nll`) and gradient R functions (`nll_grad`).

```
nll <- function(p) {
        p <- as.list(p)
        v <- do.call("nll_one", p)
        sum(v)
}
nll_grad <- function(p) {
        v <- do.call("nll_one", as.list(p))
        colSums(attr(v, "gradient"))
}
```

Finally, we can pass those functions into `optim()` with an initial vector of parameters. Here, we are careful to specify

- We are using the `"L-BFGS-B"` method so that we specify a lower bound of 0 for all parameters and an upper bound of 1 for the λ parameter
- We set the `parscale` option in the list of control parameters, which is similar to the `typsize` argument to `nlm()`. The goal here is to give `optim()` a scaling for each parameter around the optimal point.

```
x <- dat$no2
pstart <- c(5, 10, 2, 3, 0.5)
res <- optim(pstart, nll, gr = nll_grad, method = "L-BFGS-B",
             control = list(parscale = c(2, 2, 1, 1, 0.1)),
             lower = 0, upper = c(Inf, Inf, Inf, Inf, 1))
```

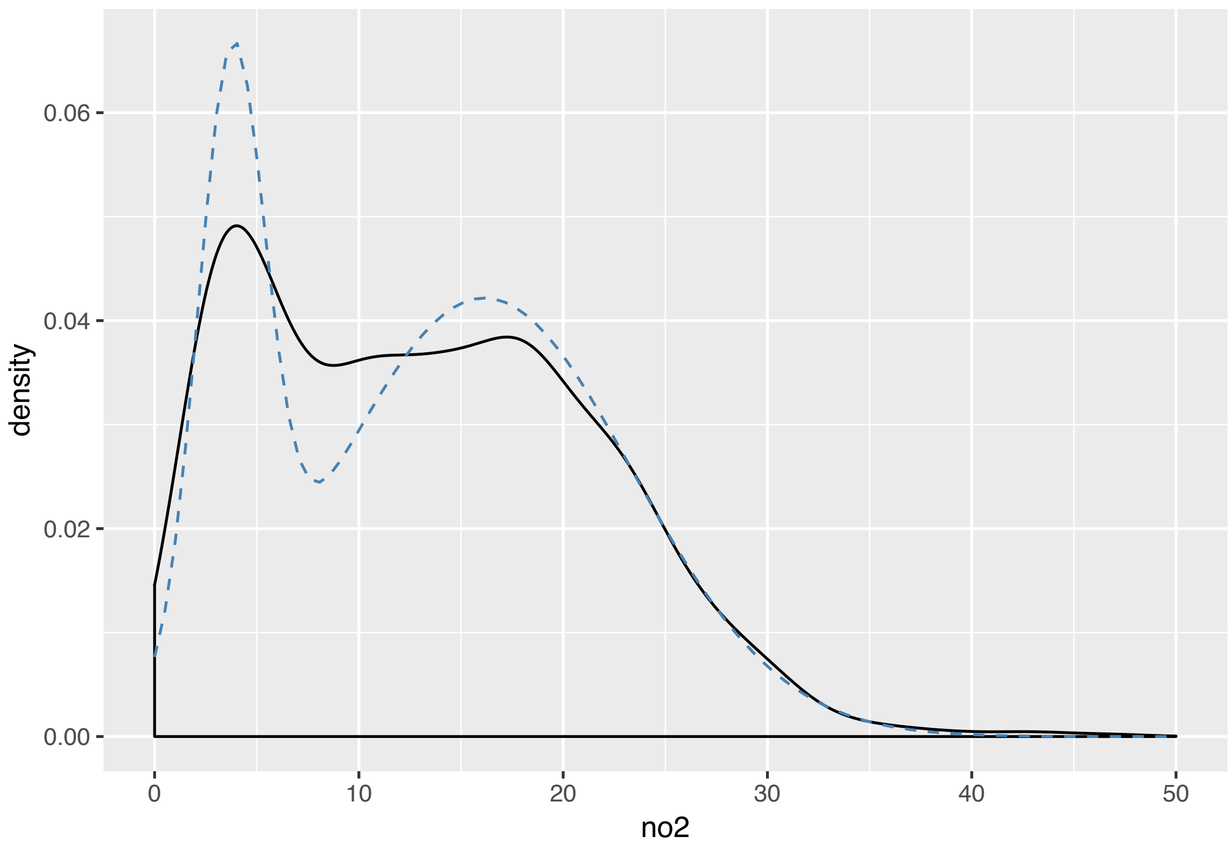
The algorithm appears to run without any warnings or messages. We can take a look at the output.

```
res
```

```
$par
[1]  3.7606598 16.1469811  1.6419640  7.2378153  0.2348927

$value
[1] 4879.924

$counts
function gradient
      17       17

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

The `convergence` code of 0 is a good sign and the parameter estimates in the `par` vector all seem reasonable. We can overlay the fitted model on to the density smooth to see how the model does.

```
xpts <- seq(0, 50, len = 100)
dens <- with(res, {
        data.frame(xpts = xpts,
                   ypts = par[5]*dnorm(xpts, par[1], par[3]) + (1-par[5])*dnorm(xpts, par[2], par[4]))
})
ggplot(dat, aes(x = no2)) +
        geom_density() +
        geom_line(aes(x = xpts, y = ypts), data = dens, col = "steelblue",
                  lty = 2)
```



The fit is still not wonderful, but at least this model captures roughly the locations of the two modes in the density. Also, it would seem that the model captures the tail of the density reasonably well, although this would need to be checked more carefully by looking at the quantiles.

Finally, as with most models and optimization schemes, it's usually a good idea to vary the starting points to see if our current estimate is a local mode.

```
pstart <- c(1, 20, 5, 2, 0.1)
res <- optim(pstart, nll, gr = nll_grad, method = "L-BFGS-B",
             control = list(parscale = c(2, 2, 1, 1, 0.1)),
             lower = 0, upper = c(Inf, Inf, Inf, Inf, 1))
res
```

```
$par
[1]  3.760571 16.146834  1.641961  7.237776  0.234892

$value
[1] 4879.924

$counts
function gradient
      22       22

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Here we see that with a slightly different starting point we get the same values and same minimum negative log-likelihood.

3.4 Conjugate Gradient

Conjugate gradient methods represent a kind of steepest descent approach "with a twist". With steepest descent, we begin our minimization of a function f starting at x_0 by traveling in the direction of the negative gradient $-f'(x_0)$. In subsequent steps, we continue to travel in the direction of the negative gradient evaluated at each successive point until convergence.

The conjugate gradient approach begins in the same manner, but diverges from steepest descent after the first step. In subsequent steps, the direction of travel must be *conjugate* to the direction most recently traveled. Two vectors u and v are conjugate with respect to the matrix A if $u'Av = 0$.

Before we go further, let's take a concrete example. Suppose we want to minimize the quadratic function

$$f(x) = \frac{1}{2}x'Ax - x'b$$

where x is a p-dimensional vector and A is a $p \times p$ symmetric matrix. Starting at a point x_0, both steepest descent and conjugate gradient would take us in the direction of $p_0 = -f'(x_0) = b - Ax_0$, which is the negative gradient. Once we have moved in that direction to the point x_1, the next direction, p_1 must satisfy $p_0'Ap_1 = 0$. So we can begin with the steepest descent direction $-f'(x_1)$ but then we must modify it to make it conjugate to p_0. The constraint $p_0'Ap_1 = 0$ allows us to back calculate the next direction, starting with $-f'(x_1)$, because we have

$$p_0'A\left(-f' - \frac{p_0'A(-f')}{p_0'Ap_0}p_0\right) = 0.$$

Without the presence of the matrix A, this is process is simply Gram-Schmidt orthogonalization. We can continue with this process, each time taking the steepest descent direction and modifying it to make it conjugate with the previous direction. The conjugate gradient process is (somewhat) interesting here because for minimizing a p-dimensional quadratic function it will converge within p steps.

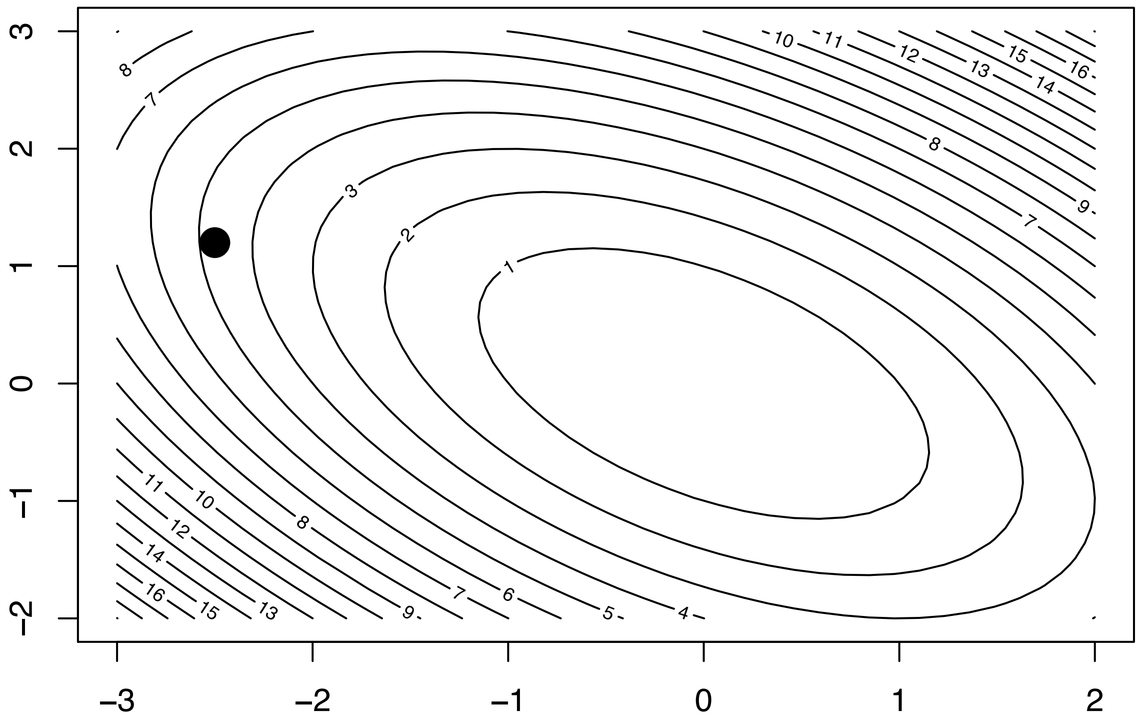
In reality, we do not deal with exactly quadratic functions and so the above-described algorithm is not feasible. However, the *nonlinear conjugate gradient method* draws from these ideas and develops a reasonable algorithm for finding the minimum of an arbitrary smooth function. It has the feature that it only requires storage of two gradient vectors, which for large problems with many parameters, is a significant savings in storage versus Newton-type algorithms which require storage of a gradient vector and a $p \times p$ Hessian matrix.

The Fletcher-Reeves nonlinear conjugate gradient algorithm works as follows. Starting with x_0,

1. Let $p_0 = -f'(x_0)$.
2. Solve
$$\min_{\alpha>0} f(x_0 + \alpha p_0)$$
for α and set $x_1 = x_0 + \alpha p_0$.
3. For $n = 1, 2, \ldots$ let $r_n = -f'(x_n)$ and $r_{n-1} = -f'(x_{n-1})$. Then set
$$\beta_n = \frac{r_n'r_n}{r_{n-1}'r_{n-1}}.$$
Finally, update $p_n = r_n + \beta_n * p_{n-1}$, and let
$$x_{n+1} = x_n + \alpha^\star p_n,$$
where $\alpha^\star$ is the solution to the problem $\min_{\alpha>0} f(x_n + \alpha p_n)$. Check convergence and if not converged, repeat.

It is perhaps simpler to describe this method with an illustration. Here, we show the contours of a 2-dimensional function.

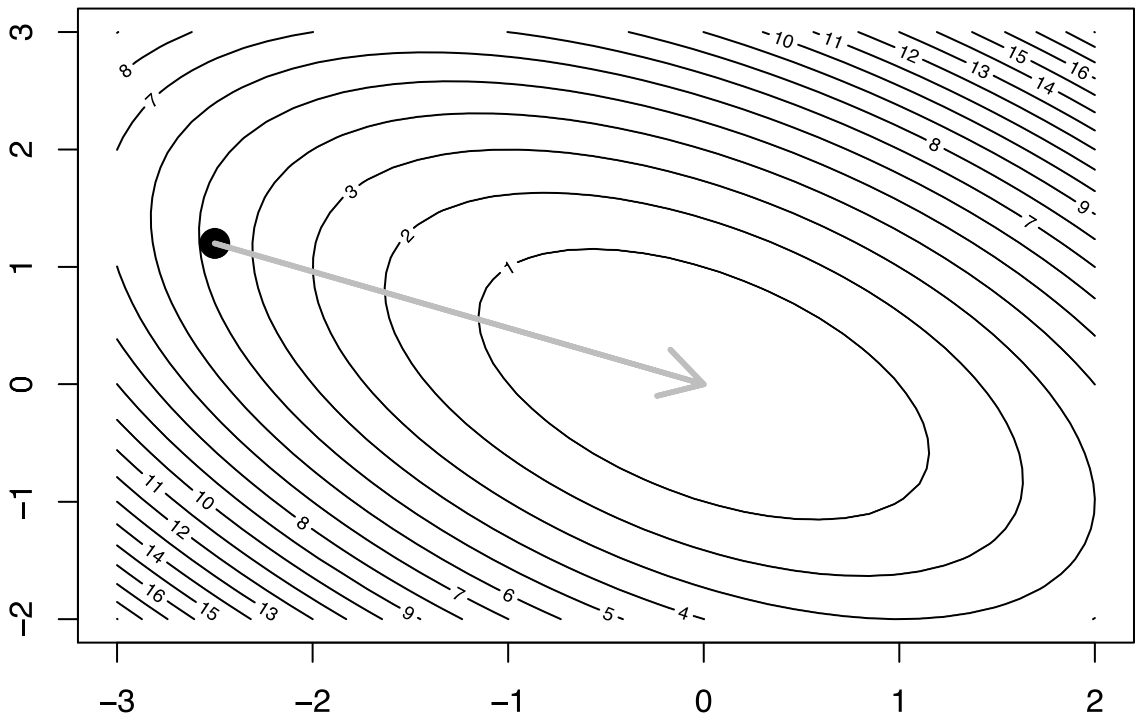
```
f <- deriv(~ x^2 + y^2 + a * x * y, c("x", "y"), function.arg = TRUE)
a <- 1
n <- 40
xpts <- seq(-3, 2, len = n)
ypts <- seq(-2, 3, len = n)
gr <- expand.grid(x = xpts, y = ypts)
feval <- with(gr, f(x, y))
z <- matrix(feval, nrow = n, ncol = n)

par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, z, nlevels = 20)
x0 <- -2.5                                  ## Initial point
y0 <- 1.2
points(x0, y0, pch = 19, cex = 2)
```



We will use as a starting point the point $(-2.5, 1.2)$, as indicated in the figure above.

From the figure, it is clear that ideally we would be able to travel in the direction that would take us directly to the minimum of the function, shown here.

```
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, z, nlevels = 20)
points(x0, y0, pch = 19, cex = 2)
arrows(x0, y0, 0, 0, lwd = 3, col = "grey")
```



If only life were so easy? The idea behind conjugate gradient is the construct that direction using a series of conjugate directions. First we start with the steepest descent direction. Here, we extract the gradient and find the optimal α value (i.e. the step size).

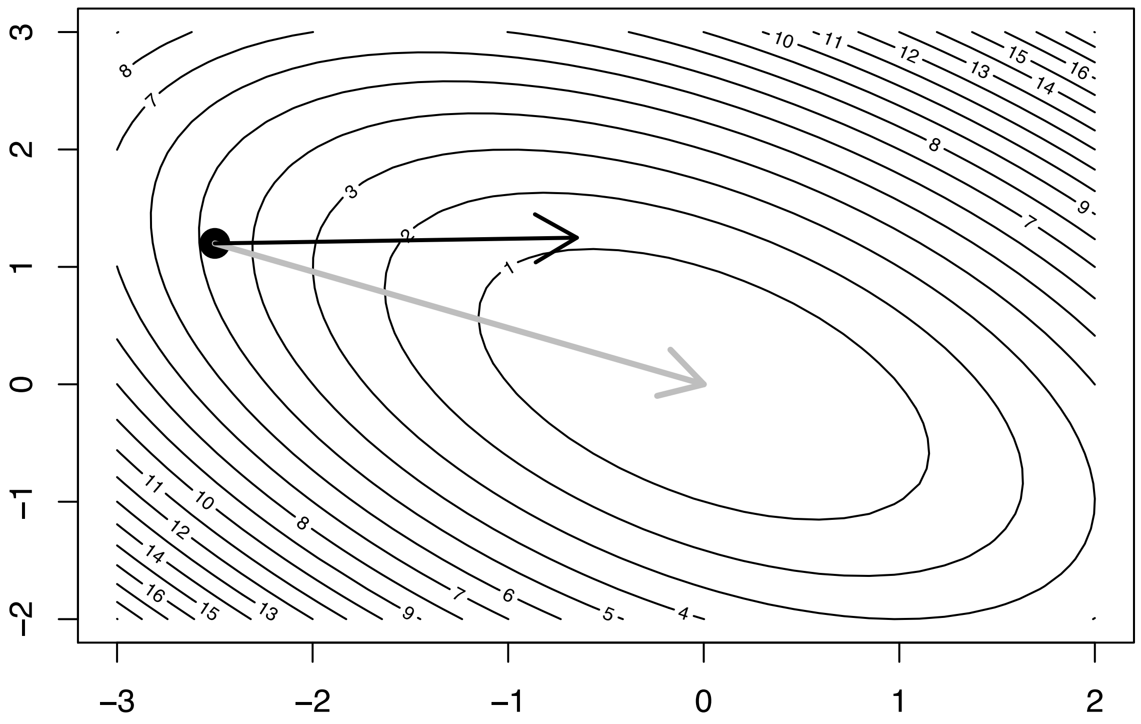
```
f0 <- f(x0, y0)
p0 <- drop(-attr(f0, "gradient"))  ## Get the gradient function
f.sub <- function(alpha) {
        ff <- f(x0 + alpha * p0[1], y0 + alpha * p0[2])
        as.numeric(ff)
}
op <- optimize(f.sub, c(0, 4))     ## Compute the optimal alpha
alpha <- op$minimum
```

Now that we've computed the gradient and the optimal α, we can take a step in the steepest descent direction

```
x1 <- x0 + alpha * p0[1]
y1 <- y0 + alpha * p0[2]
```

and plot our progress to date.

```
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, z, nlevels = 20)
points(x0, y0, pch = 19, cex = 2)
arrows(x0, y0, 0, 0, lwd = 3, col = "grey")
arrows(x0, y0, x1, y1, lwd = 2)
```

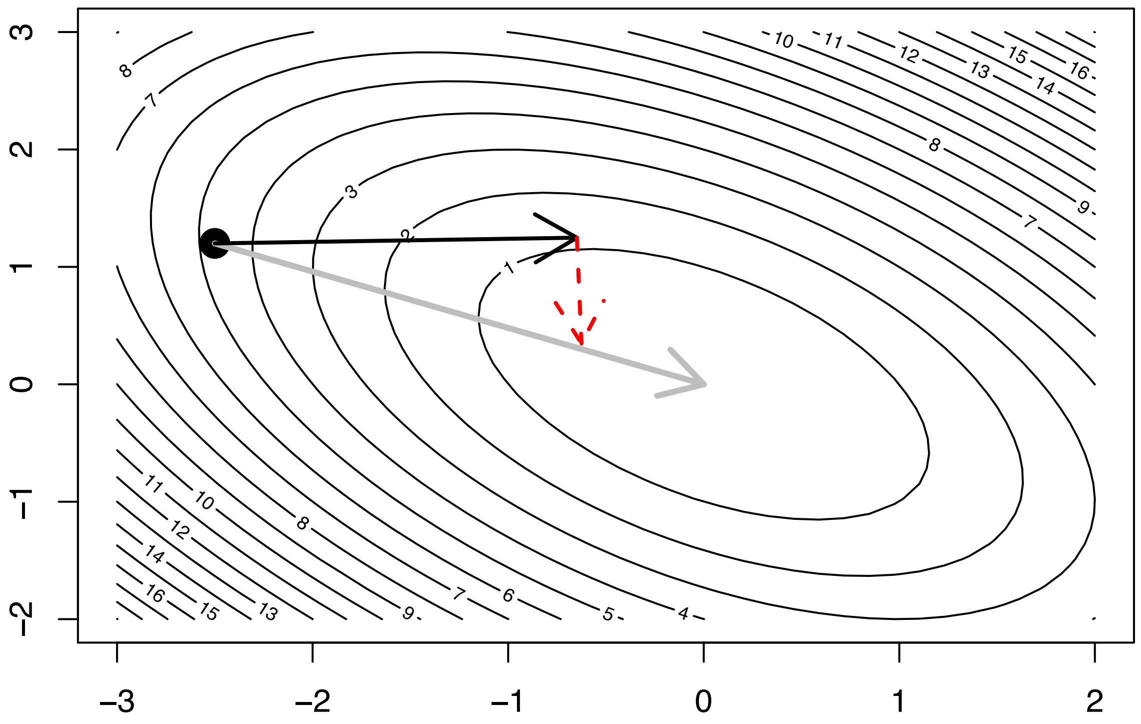


Now we need to compute the next direction in which to travel. We begin with the steepest descent direction again. The figure below shows what direction that would be (without any modifications for conjugacy).

```
f1 <- f(x1, y1)
f1g <- drop(attr(f1, "gradient"))  ## Get the gradient function
p1 <- -f1g                         ## Steepest descent direction
op <- optimize(f.sub, c(0, 4))     ## Compute the optimal alpha
alpha <- op$minimum
x2 <- x1 + alpha * p1[1]           ## Find the next point
y2 <- y1 + alpha * p1[2]
```

Now we can plot the next direction that is chosen by the usual steepest descent approach.

```
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, z, nlevels = 20)
points(x0, y0, pch = 19, cex = 2)
arrows(x0, y0, 0, 0, lwd = 3, col = "grey")
arrows(x0, y0, x1, y1, lwd = 2)
arrows(x1, y1, x2, y2, col = "red", lwd = 2, lty = 2)
```

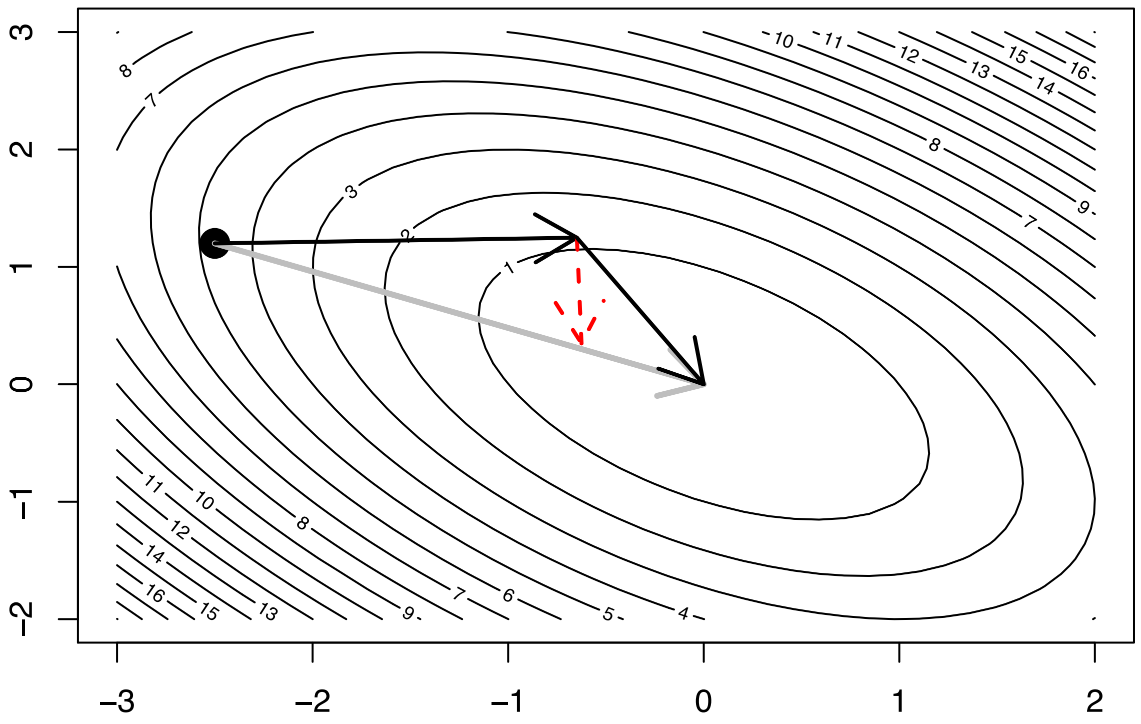


However, the conjugate gradient approach computes a slightly different direction in which to travel.

```r
f1 <- f(x1, y1)
f1g <- drop(attr(f1, "gradient"))
beta <- drop(crossprod(f1g) / crossprod(p0))  ## Fletcher-Reeves
p1 <- -f1g + beta * p0                        ## Conjugate gradient direction
f.sub <- function(alpha) {
        ff <- f(x1 + alpha * p1[1], y1 + alpha * p1[2])
        as.numeric(ff)
}
op <- optimize(f.sub, c(0, 4))                ## Compute the optimal alpha
alpha <- op$minimum
x2c <- x1 + alpha * p1[1]                     ## Find the next point
y2c <- y1 + alpha * p1[2]
```

Finally, we can plot the direction in which the conjugate gradient method takes.

```r
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, z, nlevels = 20)
points(x0, y0, pch = 19, cex = 2)
arrows(x0, y0, 0, 0, lwd = 3, col = "grey")
arrows(x0, y0, x1, y1, lwd = 2)
arrows(x1, y1, x2, y2, col = "red", lwd = 2, lty = 2)
arrows(x1, y1, x2c, y2c, lwd = 2)
```



In this case, because the target function was exactly quadratic, the process converged on the minimum in exactly 2 steps. We can see that the steepest descent algorithm would have taken many more steps to wind its way towards the minimum.

3.5 Coordinate Descent

The idea behind coordinate descent methods is simple. If f is a k-dimensional function, we can minimize f by successively minimizing each of the individual dimensions of f in a cyclic fashion, while holding the values of f in the other dimensions fixed. This approach is sometimes referred to as cyclic coordinate descent. The primary advantage of this approach is that it takes an arbitrarily complex k-dimensional problem and reduces it to a collection of k one-dimensional problems. The disadvantage is that convergence can often be painfully slow, particularly in problems where f is not well-behaved. In statistics, a popular version of this algorithm is known as *backfitting* and is used to fit generalized additive models.

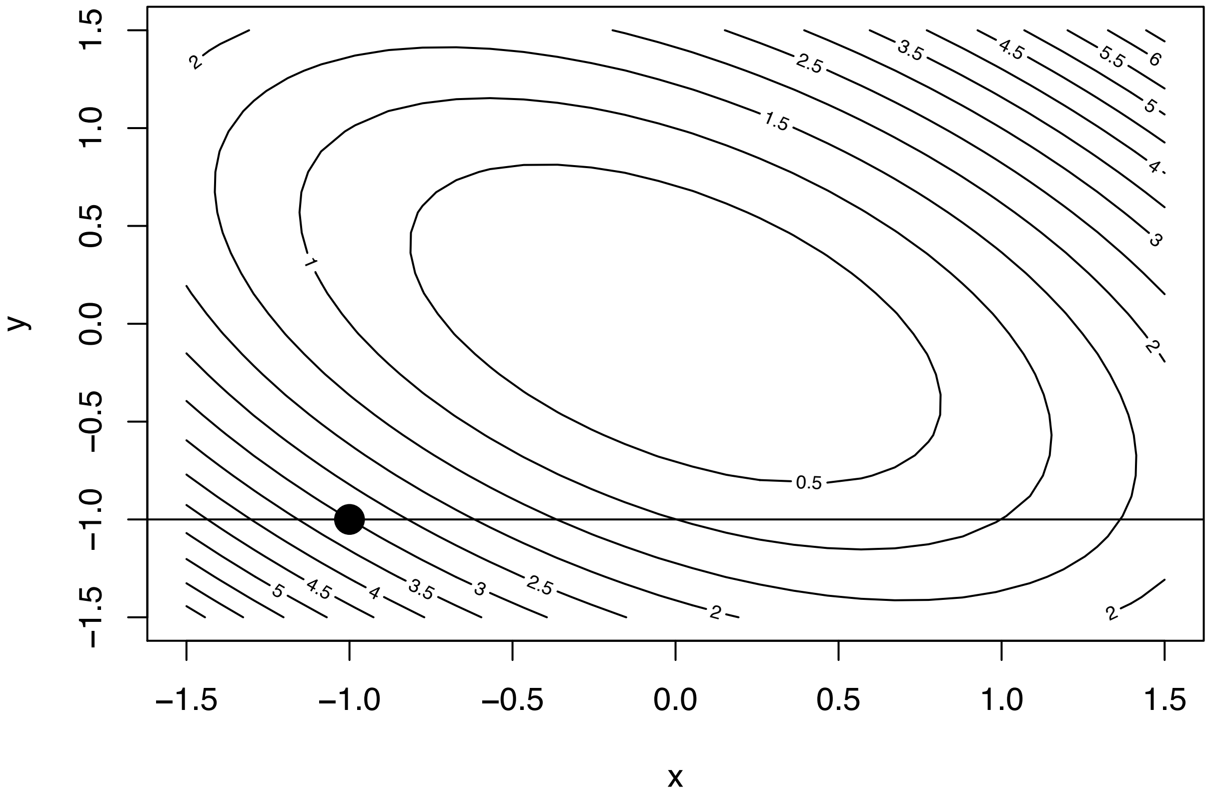
If we take a simple quadratic function we can take a detailed look at how coordinate descent works. Let's use the function

$$f(x, y) = x^2 + y^2 + xy.$$

We can make a contour plot of this function near the minimum.

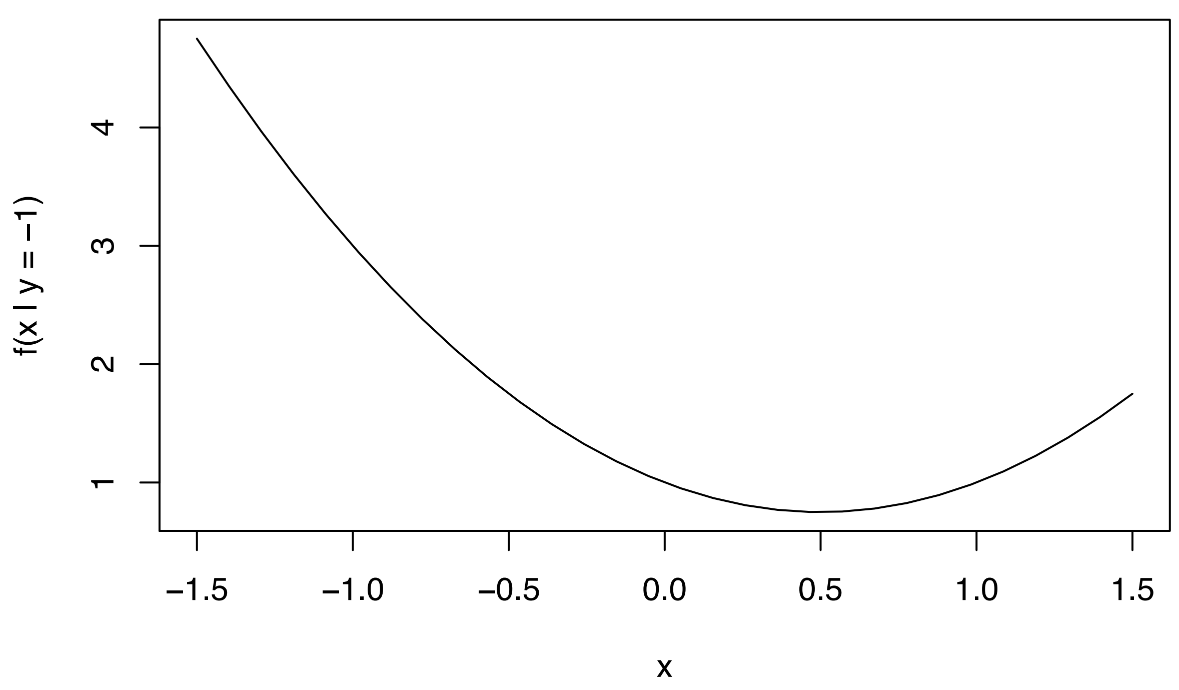
```r
f <- function(x, y) {
        x^2 + y^2 + x * y
}
n <- 30
xpts <- seq(-1.5, 1.5, len = n)
ypts <- seq(-1.5, 1.5, len = n)
gr <- expand.grid(x = xpts, y = ypts)
feval <- with(gr, matrix(f(x, y), nrow = n, ncol = n))
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, feval, nlevels = 20, xlab = "x", ylab = "y")
```

```r
points(-1, -1, pch = 19, cex = 2)
abline(h = -1)
```



Let's take as our initial point $(-1, -1)$ and begin our minimization along the x dimension. We can draw a transect at the $y = -1$ level (thus holding y constant) and attempt to find the minimum along that transect. Because f is a quadratic function, the one-dimensional function induced by holding $y = -1$ is also a quadratic.

```r
feval <- f(xpts, y = -1)
plot(xpts, feval, type = "l", xlab = "x", ylab = "f(x | y = -1)")
```



We can minimize this one-dimensional function with the `optimize()` function (or we could do it by hand if we're not lazy).

```
fx <- function(x) {
        f(x, y = -1)
}
op <- optimize(fx, c(-1.5, 1.5))
op
```

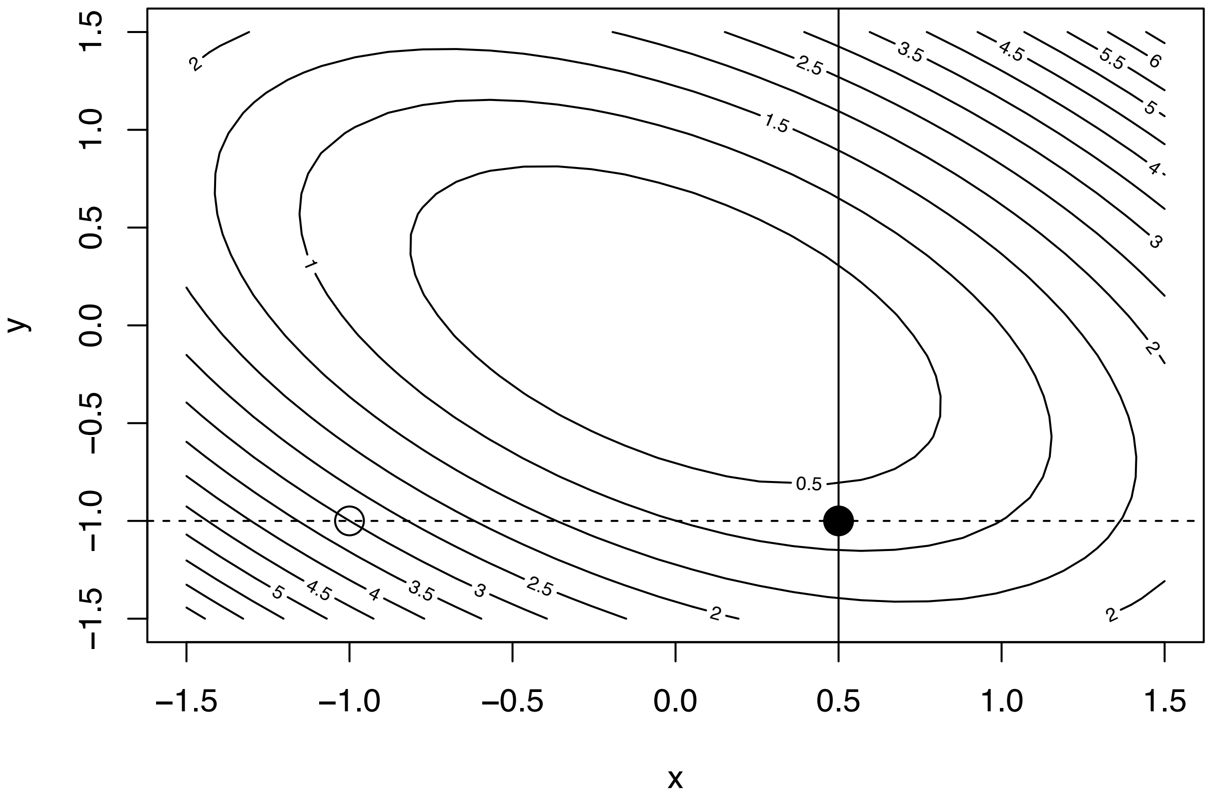
```
$minimum
[1] 0.5

$objective
[1] 0.75
```

Granted, we could have done this analytically because we are looking at a simple quadratic function. But in general, you will need a one-dimensional optimizer (like the `optimize()` function in R) to complete each of the coordinate descent iterations.

This completes one iteration of the coordinate descent algorithm and our new starting point is $(0.5, -1)$. Let's store this new x value and move on to the next iteration, which will minimize along the y direction.

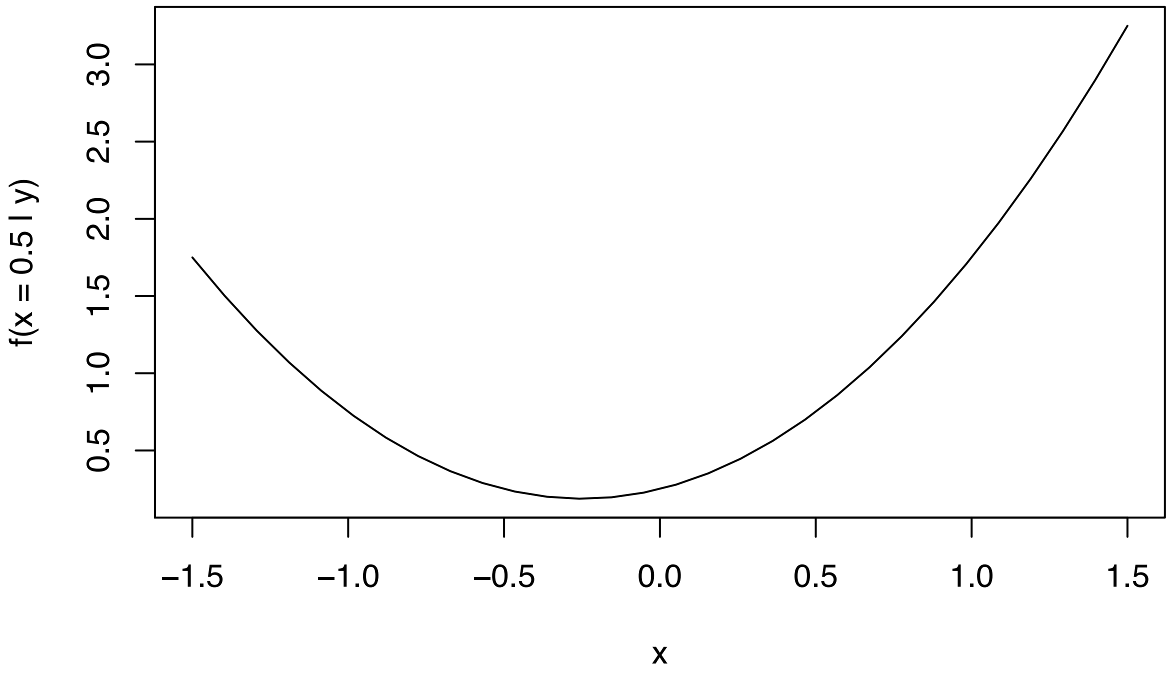
```
x1 <- op$minimum
```

```
feval <- with(gr, matrix(f(x, y), nrow = n, ncol = n))
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, feval, nlevels = 20, xlab = "x", ylab = "y")
points(-1, -1, pch = 1, cex = 2)      ## Initial point
abline(h = -1, lty = 2)
points(x1, -1, pch = 19, cex = 2)     ## After one step
abline(v = x1)
```



The transect drawn by holding $x = 0.5$ is shown in the Figure above. The one-dimensional function corresponding to that transect is shown below (again, a one-dimensional quadratic function).

```
feval <- f(x = x1, ypts)
plot(xpts, feval, type = "l", xlab = "x",
     ylab = sprintf("f(x = %.1f | y)", x1))
```



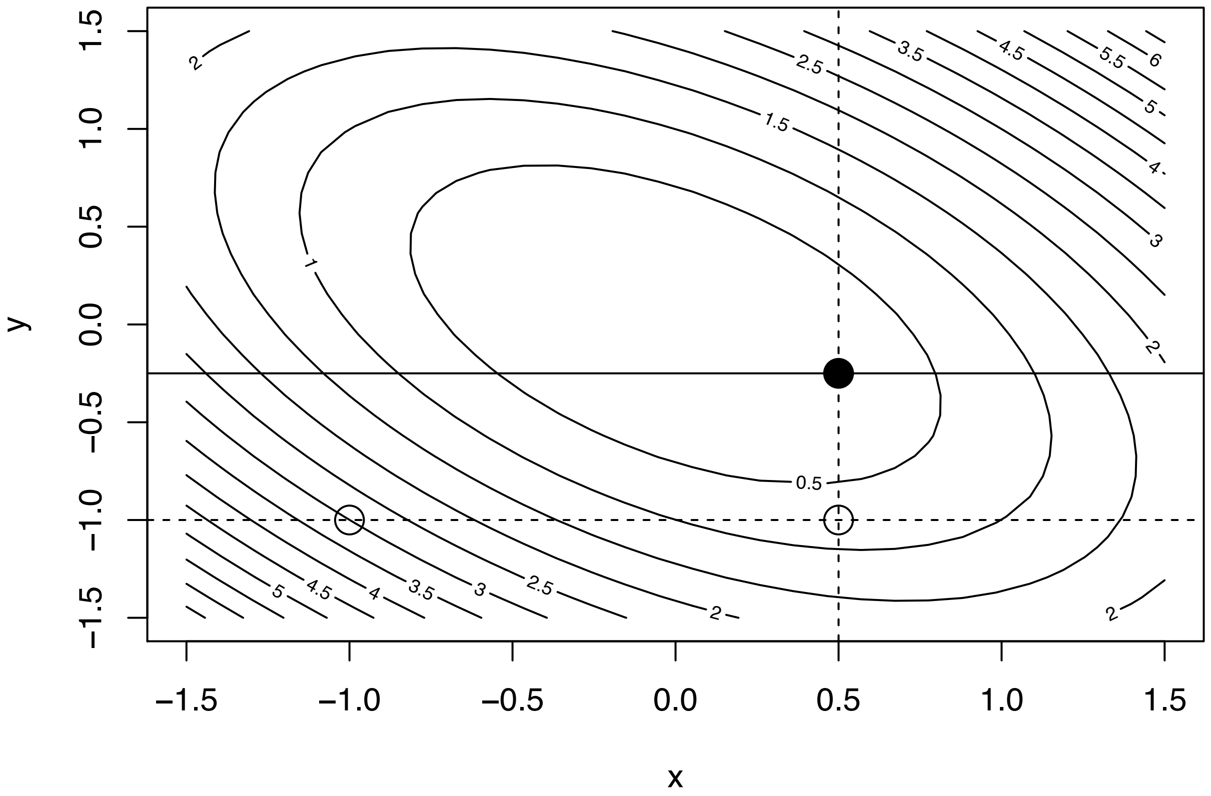
Minimizing this one-dimensional function, we get the following.

```
fy <- function(y) {
        f(x = x1, y)
}
op <- optimize(fy, c(-1.5, 1.5))
op
```

```
$minimum
[1] -0.25

$objective
[1] 0.1875
```

This completes another iteration of the coordinate descent algorithm and we can plot our progress below.

```
y1 <- op$minimum
feval <- with(gr, matrix(f(x, y), nrow = n, ncol = n))
par(mar = c(5, 4, 1, 1))
contour(xpts, ypts, feval, nlevels = 20, xlab = "x", ylab = "y")
points(-1, -1, pch = 1, cex = 2)   ## Initial point
abline(h = -1, lty = 2)
points(x1, -1, pch = 1, cex = 2)   ## After one step
abline(v = x1, lty = 2)
points(x1, y1, pch = 19, cex = 2)  ## After two steps
abline(h = y1)                     ## New transect
```



We can see that after two iterations we are quite a bit closer to the minimum. But we still have a ways to go, given that we can only move along the coordinate axis directions. For a truly quadratic function, this is not an efficient way to find the minimum, particularly when Newton's method will find the minimum in a single step! Of course, Newton's method can achieve that kind of performance because it uses two derivatives worth of information. The coordinate descent approach uses no derivative information. There's no free lunch!

In the above example, the coordinates x and y were moderately correlated but not dramatically so. In general, coordinate descent algorithms show very poor performance when the coordinates are strongly correlated.

The specifics of the coordinate descent algorithm will vary greatly depending on the general function being minimized, but the essential algorithm is as follows. Given a function $f : \mathbb{R}^p \to \mathbb{R}$,

1. For $j = 1, \dots, p$, minimize $f_j(x) = f(\dots, x_{j-1}, x, x_{j+1}, \dots)$ wbere $x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_p$ are all held fixed at their current values. For this use any simple one-dimensional optimizer.
2. Check for convergence. If not converged, go back to 1.

3.5.1 Convergence Rates

To take a look at the convergence rate for coordinate descent, we will use as an example, a slightly more general version of the quadratic function above,

$$f(x, y) = x^2 + y^2 + axy,$$

where here, a, represents the amount of correlation or coupling between the x and y coordinates. If $a = 0$ there is no coupling and x and y vary independently.

At each iteration of the coordinate descent algorithm, we minimize a one-dimensional version of this function. If we fix $y = c$, then we want to minimize

$$f_{y=c}(x) = x^2 + c^2 + acx.$$

Taking the derivative of this with respect to x and setting it equal to zero gives us the minimum at

$$x_{min} = \frac{-ac}{2}$$

Similarly, if we fix $x = c$, then we can minimize an analogous function $f_{x=c}(y)$ to get a minimum point of $y_{min} = \frac{-ac}{2}$.

Looking at the coordinate descent algorithm, we can develop the recurrence relationship

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} -\frac{a}{2} y_n \\ -\frac{a}{2} x_{n+1} \end{pmatrix}$$

Rewinding this back to the inital point, we can then write that

$$|x_n - x_0| = |x_n - 0| = \left(\frac{a}{2}\right)^{2n-1} y_0.$$

where x_0 is the minimum point along the x direction. We can similarly say that

$$|y_n - y_0| = \left(\frac{a}{2}\right)^{2n} x_0.$$

Looking at the rates of convergence separately for each dimension, we can then show that in the x direction,

$$\frac{|x_{n+1} - x_0|}{|x_n - x_0|} = \frac{\left(\frac{a}{2}\right)^{2(n+1)-1} y_0}{\left(\frac{a}{2}\right)^{2n-1} y_0} = \left(\frac{a}{2}\right)^2.$$

In order to achieve linear convergence for this algorithm, we must have $\left(\frac{a}{2}\right)^2 \in (0, 1)$, which can be true for some values of a. But for values of $a \geq 2$ we would not even be able to obtain linear convergence.

In summary, coordinate descent algorithms are conceptually (and computationally) easy to implement but depending on the nature of the target function, convergence may not be possible, even under seemingly reasonable scenarios like the simple one above. Given that we typically do not have very good information about the nature of the target function, particularly in high-dimensional problems, coordinate descent algorithms should be used with care.

3.5.2 Generalized Additive Models

Before we begin this section, I want to point out that Brian Caffo has a nice video introduction to generalized additive models on his YouTube channel.

Generalized additive models represent an interesting class of models that provide nonparametric flexibility to estimate a high-dimensional function without succumbing to the notorious "curse of dimensionality". In the traditional linear model, we model the outcome y as

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 \cdots + \beta_p x_p + \varepsilon.$$

Generalized additive models replace this formulation with a slightly more general one,

$$y = \alpha + s_1(x_1 \mid \lambda_1) + s_2(x_2 \mid \lambda_2) + \cdots + s_p(x_p \mid \lambda_p) + \varepsilon$$

where $s_1, \ldots, s_p$ are *smooth functions* whose smoothness is controled by the parameters $\lambda_1, \ldots, \lambda_p$. The key compromise of this model is that rather than estimate an arbitrary smooth p-dimensional function, we estimate a series of p one-dimensional functions. This is a much simpler problem but still allows us to capture a variety of nonlinear relationships.

The question now is how do we estimate these smooth functions? Hastie and Tibshirani proposed a *backfitting algorithm* whereby each $s_j()$ would be estimated one-at-a-time while holding all of the other functions constant. This is essentially a coordinate descent algorithm where the coordinates are one-dimensional functions in a function space.

The $s_j()$ functions can be estimated using any kind of smoother. Hastie and Tibshirani used running median smoothers for robustness in many of their examples, but one could use splines, kernel smoothers, or many others.

The backfitting algorithm for additive models works as follows. Given a model of the form

$$y_i = \alpha + \sum_{j=1}^{p} s_j(x_{ij} \mid \lambda_j) + \varepsilon_i$$

where $i = 1, \ldots, n$, 1. Initialize $\alpha = \frac{1}{n}\sum_{i=1}^{n} y_i$, $s_1 = s_2 = \cdots = s_p = 0$.

2. Given current values $s_1^{(n)}, \ldots, s_p^{(n)}$, for $j = 1, \ldots, p$, Let

   $$r_{ij} = y_i - \alpha - \sum_{\ell \neq j} s_\ell(x_{i\ell} \mid \lambda_\ell)$$

   so that r_{ij} is the partial residual for predictor j and observation i. Given this set of partial residuals $r_{1j}, \ldots, r_{nj}$, we can estimate s_j by smoothing the relationship between r_{ij} and x_{ij} using any smoother we choose. Essentially, we need to solve the mini-problem

   $$r_{ij} = s_j(x_{ij} \mid \lambda_j) + \varepsilon_i$$

   using standard nonparametric smoothers. As part of this process, we may need to estimate λ_j using a procedure like generalized cross-validation or something similar. At the end of this step we have $s_1^{(n+1)}, \ldots, s_p^{(n+1)}$

3. We can evaluate

   $$\Delta = \sum_{j=1}^{p} \left\| s_j^{(n+1)} - s_j^{(n)} \right\|$$

   or

   $$\Delta = \frac{\sum_{j=1}^{p} \left\| s_j^{(n+1)} - s_j^{(n)} \right\|}{\sum_{j=1}^{p} \left\| s_j^{(n)} \right\|}.$$

   where $\|\cdot\|$ is some reasonable metric. If Δ is less than some pre-specificed tolerance, we can stop the algorithm. Otherwise, we can go back to Step 2 and do another round of backfitting.

4 The EM Algorithm

The EM algorithm is one of the most popular algorithms in all of statistics. A quick look at Google Scholar shows that the paper by Art Dempster, Nan Laird, and Don Rubin has been cited more than 50,000 times. The EM stands for "Expectation-Maximization", which indicates the two-step nature of the algorithm. At a high level, there are two steps: The "E-Step" and the "M-step" (duh!).

The EM algorithm is not so much an algorithm as a methodology for creating a family of algorithms. We will get into how exactly it works a bit later, but suffice it to say that when someone says "We used the EM algorithm," that probably isn't enough information to understand exactly what they did. The devil is in the details and most problems will need a bit of hand crafting. That said, there are a number of canonical problems now where an EM-type algorithm is the standard approach.

The basic idea underlying the EM algorithm is as follows. We *observe* some data that we represent with Y. However, there are some *missing* data, that we represent with Z, that make life difficult for us. Together, the observed data Y and the missing data Z make up the *complete* data $X = (Y, Z)$.

1. We imagine the complete data have a density $g(y, z \mid \theta)$ that is parametrized by the vector of parameters θ. Because of the missing data, we cannot evaluate g.
2. The observed data have the density
$$f(y \mid \theta) = \int g(y, z \mid \theta)\, dz$$
and the *observed data log-likelihood* is $\ell(\theta \mid y) = \log f(y \mid \theta)$.
3. The problem now is that $\ell(\theta \mid y)$ is difficult to evaluate or maximize because of the integral (for discrete problems this will be a sum). However, in order to estimate θ via maximum likelihood *using only the observed data*, we need to be able to maximize $\ell(\theta \mid y)$.
4. The complete data density usually has some nice form (like being an exponential family member) so that if we had the missing data Z, we could easily evaluate $g(y, z \mid \theta)$.

Given this setup, the basic outline of the EM algorithm works as follows:

1. E-step: Let θ_0 be the current estimate of θ. Define
$$Q(\theta \mid \theta_0) = \mathbb{E}\left[\log g(y, z \mid \theta) \mid y, \theta_0\right]$$
2. M-step: Maximize $Q(\theta \mid \theta_0)$ with respect to θ to get the next value of θ.
3. Goto 1 unless converged.

In the E-step, the expectation is taken with respect to the *missing data density*, which is
$$h(z \mid y, \theta) = \frac{g(y, z \mid \theta)}{f(y \mid \theta)}.$$

Because we do not know θ, we can plug in θ_0 to evaluate the missing data density. In particular, one can see that it's helpful if the $\log g(y, z \mid \theta)$ is linear in the missing data so that taking the expectation is a simple operation.

4.1 EM Algorithm for Exponential Families

Data that are generated from a *regular exponential family* distribution have a density that takes the form
$$g(x \mid \theta) = h(x) \exp(\theta' t(x))/a(\theta).$$

where θ is the canonical parameter and $t(x)$ is the vector of sufficient statistics. When thinking about the EM algorithm, the idea scenario is that the *complete data density* can be written as an exponential family. In that case, for the E-step, if y represents the observed component of the complete data, we can write

$$\begin{aligned} Q(\theta \mid \theta_0) &= \mathbb{E}[\log g(x \mid \theta) \mid y, \theta_0] \\ &= \log h(x) - \theta' \mathbb{E}[t(x) \mid y, \theta_0] - \log a(\theta) \end{aligned}$$

(Note: We can ignore the $h(x)$ term because it does not involve the θ parameter.) In order to maximize this function with respect to θ, we can take the derivative and set it equal to zero,

$$Q'(\theta \mid \theta_0) = \mathbb{E}[t(x) \mid y, \theta_0] - \mathbb{E}_\theta[t(x)] = 0.$$

Hence, for exponential family distributions, executing the M-step is equivalent to setting

$$\mathbb{E}[t(x) \mid y, \theta_0] = \mathbb{E}_\theta[t(x)]$$

where $\mathbb{E}_\theta[t(x)]$ is the unconditional expectation of the complete data and $\mathbb{E}[t(x) \mid y, \theta_0]$ is the conditional expectation of the missing data, given the observed data.

4.2 Canonical Examples

In this section, we give some canonical examples of how the EM algorithm can be used to estimate model parameters. These examples are simple enough that they can be solved using more direct methods, but they are nevertheless useful for demonstrating how to set up the two-step EM algorithm in various scenarios.

4.2.1 Two-Part Normal Mixture Model

Suppose we have data $y_1, \dots, y_n$ that are sampled independently from a two-part mixture of Normals model with density

$$f(y \mid \theta) = \lambda\varphi(y \mid \mu_1, \sigma_1^2) + (1 - \lambda)\varphi(y \mid \mu_2, \sigma_2^2).$$

where $\varphi(y \mid \mu, \sigma^2)$ is the Normal density with mean μ and variance σ^2. The unknown parameter vector is $\theta = (\mu_1, \mu_2, \sigma_1^2, \sigma_2^2, \lambda)$ and the log-likelihood is

$$\log f(y_1, \dots, y_n \mid \theta) = \log \sum_{i=1}^n \lambda\varphi(y_i \mid \mu_1, \sigma_1) + (1 - \lambda)\varphi(y_i \mid \mu_2, \sigma_2).$$

This problem is reasonably simple enough that it could be solved using a direct optimization method like Newton's method, but the EM algorithm provides a nice stable approach to finding the optimum.

The art of applying the EM algorithm is coming up with a useful complete data model. In this example, the approach is to hypothesize that each observation comes from one of two populations parameterized by (μ_1, σ_1^2) and (μ_2, σ_2^2), respectively. The "missing data" in this case are the labels identifying which observation came from which population. Therefore, we assert that there are missing data $z_1, \dots, z_n$ such that

$$z_i \sim \text{Bernoulli}(\lambda).$$

When $z_i = 1$, y_i comes from population 1 and when $z_i = 0$, y_i comes from population 2.

The idea is then that the data are sampled in two stages. First we sample z_i to see which population the data come from and then given z_i, we can sample y_i from the appropriate Normal distribution. The joint density of the observed and missing data, i.e. the complete data density, is then

$$g(y, z \mid \theta) = \varphi(y \mid \mu_1, \sigma_1^2)^z \varphi(y \mid \mu_2, \sigma_2^2)^{1-z} \lambda^z (1 - \lambda)^{1-z}.$$

It's easy to show that

$$\sum_{z=0}^{1} g(y, z \mid \theta) = f(y \mid \theta)$$

so that when we "integrate" out the missing data, we get the observed data density.

The complete data log-likelihood is then

$$\log g(y, z \mid \theta) = \sum_{i=1}^{n} z_i \log \varphi(y_i \mid \mu_1, \sigma_1^2) + (1 - z_i) \log \varphi(y_i \mid \mu_2, \sigma_2^2) + z_i \log \lambda + (1 - z_i) \log(1 - \lambda).$$

Note that this function is nice and linear in the missing data z_i. To evaluate the $Q(\theta \mid \theta_0)$ function we need to take the expectation of the above expression with respect to the missing data density $h(z \mid y, \theta)$. But what is that? The missing data density will be proportional to the complete data density, so that

$$\begin{aligned} h(z \mid y, \theta) &\propto \varphi(y \mid \mu_1, \sigma_1^2)^z \varphi(y \mid \mu_2, \sigma_2^2)^{1-z} \lambda^z (1 - \lambda)^{1-z} \\ &= (\lambda \varphi(y \mid \mu_1, \sigma_1^2))^z ((1 - \lambda) \varphi(y \mid \mu_2, \sigma_2^2))^{1-z} \\ &= \text{Bernoulli}\left(\frac{\lambda \varphi(y \mid \mu_1, \sigma_1^2)}{\lambda \varphi(y \mid \mu_1, \sigma_1^2) + (1 - \lambda) \varphi(y \mid \mu_2, \sigma_2^2)} \right) \end{aligned}$$

From this, what we need to compute the $Q()$ function is $\pi_i = \mathbb{E}[z_i \mid y_i, \theta_0]$. Given that, wen then compute the $Q()$ function in the E-step.

$$\begin{aligned} Q(\theta \mid \theta_0) &= \mathbb{E}\left[\sum_{i=1}^{n} z_i \log \varphi(y \mid \mu_1, \sigma_1^2) + (1 - z_i) \log \varphi(y \mid \mu_2, \sigma_2^2) + z_i \log \lambda + (1 - z_i) \log(1 - \lambda) \right] \\ &= \sum_{i=1}^{n} \pi_i \log \varphi(y \mid \mu_1, \sigma_1^2) + (1 - \pi_i) \varphi(y \mid \mu_2, \sigma_2^2) + \pi_i \log \lambda + (1 - \pi_i) \log(1 - \lambda) \\ &= \sum_{i=1}^{n} \pi_i \left[-\frac{1}{2} \log 2\pi\sigma_1^2 - \frac{1}{2\sigma_1^2}(y_i - \mu_1)^2 \right] + (1 - \pi_i) \left[-\frac{1}{2} \log 2\pi\sigma_2^2 - \frac{1}{2\sigma_2^2}(y_i - \mu_2)^2 \right] \\ &\quad + \pi_i \log \lambda + (1 - \pi_i) \log(1 - \lambda) \end{aligned}$$

In order to compute π_i, we will need to use the current estimates of μ_1, σ_1^2, μ_2, and σ_2^2 (in addition to the data $y_1, \ldots, y_n$). We can then compute the gradient of Q in order maximize it for the current iteration. After doing that we get the next values, which are

$$\begin{aligned} \hat{\mu}_1 &= \frac{\sum \pi_i y_i}{\sum \pi_i} \\ \hat{\mu}_2 &= \frac{\sum (1 - \pi_i) y_i}{\sum 1 - \pi_i} \\ \hat{\sigma}_1^2 &= \frac{\sum \pi_i (y_i - \mu_1)^2}{\sum \pi_i} \\ \hat{\sigma}_2^2 &= \frac{\sum (1 - \pi_i)(y_i - \mu_2)^2}{\sum (1 - \pi_i)} \\ \hat{\lambda} &= \frac{1}{n} \sum \pi_i \end{aligned}$$

Once we have these updated estimates, we can go back to the E-step and recompute our Q function.

4.2.2 Censored Exponential Data

Suppose we have survival times $x_1, \dots, x_n \sim$ Exponential(λ). However, we do not observe these survival times because some of them are censored at times $c_1, \dots, c_n$. Because the censoring times are known, what we actually observe are the data $(\min(y_1, c_1), \delta_1), \dots, (\min(y_n, c_n), \delta_n)$, where $\delta = 1$ if $y_i \le c_i$ and $\delta = 0$ if y_i is censored at time c_i.

The complete data density is simply the exponential distribution with rate parameter λ,

$$g(x_1, \dots, x_n \mid \lambda) = \prod_{i=1}^{n} \frac{1}{\lambda} \exp(-x_i/\lambda).$$

To do the E-step, we need to compute

$$Q(\lambda \mid \lambda_0) = \mathbb{E}[\log g(x_1, \dots, x_n \mid \lambda) \mid \mathbf{y}, \lambda_0]$$

We can divide the data into the observations that we fully observe ($\delta_i = 1$) and those that are censored ($\delta_i = 0$). For the censored data, their complete survival time is "missing", so can denote the complete survival time as z_i. Given that, the $Q(\lambda \mid \lambda_0)$ function is

$$Q(\lambda \mid \lambda_0) = \mathbb{E}\left\{ -n \log \lambda - \frac{1}{\lambda}\left[\sum_{i=1}^{n} \delta_i y_i + (1-\delta_i) z_i.\right] \middle| \mathbf{y}, \lambda_0 \right\}$$

But what is $\mathbb{E}[z_i \mid y_i, \lambda_0]$? Because we assume the underlying data are exponentially distributed, we can use the "memoryless" property of the exponential distribution. That is, given that we have survived until the censoring time c_i, our expected survival time beyond that is simply λ. Because we don't know λ yet we can plug in our current best estimate. Now, for the E-step we have

$$Q(\lambda \mid \lambda_0) = -n \log \lambda - \frac{1}{\lambda}\left[\sum_{i=1}^{n} \delta_i y_i + (1-\delta_i)(c_i + \lambda_0)\right]$$

With the Q function removed of missing data, we can execute the M-step and maximize the above function to get

$$\hat{\lambda} = \frac{1}{n}\left[\sum_{i=1}^{n} \delta_i y_i + (1-\delta_i)(c_i + \lambda_0)\right]$$

We can then update $\lambda_0 = \hat{\lambda}$ and go back and repeat the E-step.

4.3 A Minorizing Function

One of the positive qualities of the EM algorithm is that it is very stable. Unlike Newton's algorithm, where each iteration may or may not be closer to the optimal value, each iteratation of the EM algorithm is designed to increase the observed log-likelihood. This is the *ascent property of the EM algorithm*, which we will show later. This stability, though, comes at a price—the EM algorithm's convergence rate is linear (while Newton's algorithm is quadratic). This can make running the EM algorithm painful at times, particularly when one has to compute standard errors via a resampling approach like the bootstrap.

The EM algorithm is a *minorization* approach. Instead of directly maximizing the log-likelihood, which is difficult to evaluate, the algorithm constructs a minorizing function and optimizes that function instead. What is a minorizing function? Following Chapter 7 of Jan de Leeuw's *Block Relaxation Algorithms in Statistics* a function g *minorizes* f over $\mathcal{X}$ at y if

1. $g(x) \le f(x)$ for all $x \in \mathcal{X}$
2. $g(y) = f(y)$

In the description of the EM algorithm above, $Q(\theta \mid \theta_0)$ is the minorizing function. The benefits of this approach are

1. The $Q(\theta \mid \theta_0)$ is a much nicer function that is easy to optimize
2. Because the $Q(\theta \mid \theta_0)$ minorizes $\ell(\theta \mid y)$, maximizing it is guaranteed to increase (or at least not decrease) $\ell(\theta \mid y)$. This is because if θ_n is our current estimate of θ and $Q(\theta \mid \theta_n)$ minorizes $\ell(\theta \mid y)$ at θ_n, then we have
$$\ell(\theta_{n+1} \mid y) \geq Q(\theta_{n+1} \mid \theta_n) \geq Q(\theta_n \mid \theta_n) = \ell(\theta_n \mid y).$$

Let's take a look at how this minorization process works. We can begin with the observe log-likelihood

$$\log f(y \mid \theta) = \log \int g(y, z \mid \theta)\, dz.$$

Using the time-honored strategy of adding and subtracting, we can show that if θ_0 is our current estimate of θ,

$$\begin{aligned}
\log f(y \mid \theta) - \log f(y \mid \theta_0) &= \log \int g(y, z \mid \theta)\, dz - \log \int g(y, z \mid \theta_0)\, dz \\
&= \log \frac{\int g(y, z \mid \theta)\, dz}{\int g(y, z \mid \theta_0)\, dz} \\
&= \log \frac{\int g(y, z \mid \theta_0) \frac{g(y,z \mid \theta)}{g(y,z \mid \theta_0)}\, dz}{\int g(y, z \mid \theta_0)\, dz}
\end{aligned}$$

Now, because we have defined

$$h(z \mid y, \theta) = \frac{g(y, z \mid \theta)}{f(y \mid \theta)} = \frac{g(y, z \mid \theta)}{\int g(y, z \mid \theta)\, dz}$$

we can write

$$\begin{aligned}
\log f(y \mid \theta) - \log f(y \mid \theta_0) &= \log \int h(z \mid y, \theta_0) \frac{g(y, z \mid \theta)}{g(y, z \mid \theta_0)}\, dz \\
&= \log \mathbb{E}\left[\left. \frac{g(y, z \mid \theta)}{g(y, z \mid \theta_0)} \right| y, \theta_0 \right]
\end{aligned}$$

Because the log function is concave, Jensen's inequality tells us that

$$\log \mathbb{E}\left[\left. \frac{g(y, z \mid \theta)}{g(y, z \mid \theta_0)} \right| y, \theta_0 \right] \geq \mathbb{E}\left[\log \left. \frac{g(y, z \mid \theta)}{g(y, z \mid \theta_0)} \right| y, \theta_0 \right].$$

Taking this, we can then write

$$\log f(y \mid \theta) - \log f(y \mid \theta_0) \geq \mathbb{E}\left[\log \left. \frac{g(y, z \mid \theta)}{g(y, z \mid \theta_0)} \right| y, \theta_0 \right],$$

which then gives us

$$\begin{aligned}
\log f(y \mid \theta) &\geq \log f(y \mid \theta_0) + \mathbb{E}[\log g(y, z \mid \theta) \mid y, \theta_0] - \mathbb{E}[\log g(y, z \mid \theta_0) \mid y, \theta_0] \\
&= \log f(y \mid \theta_0) + Q(\theta \mid \theta_0) - Q(\theta_0 \mid \theta_0)
\end{aligned}$$

The right-hand side of the above equation, the middle part of which is a function of θ, is our minorizing function. We can see that for $\theta = \theta_0$ we have that the minorizing function is equal to $\log f(y \mid \theta_0)$.

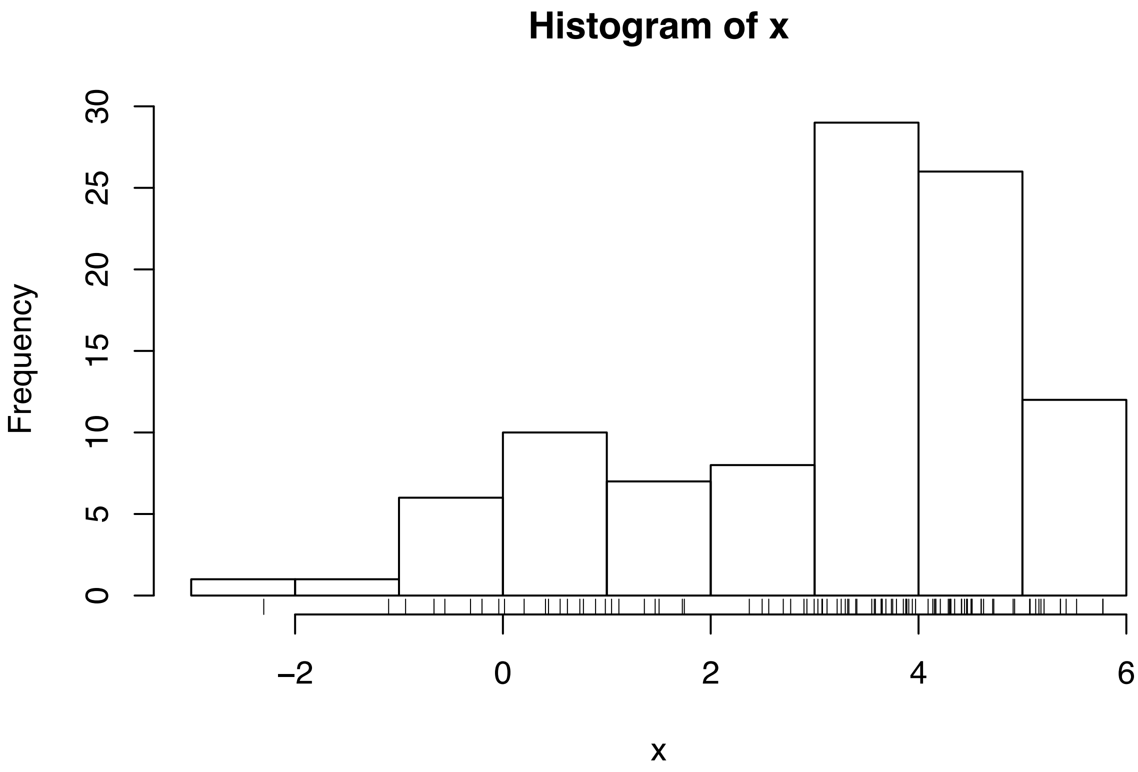
4.3.1 Example: Minorization in a Two-Part Mixture Model

We will revisit the two-part Normal mixture model from before. Suppose we have data $y_1, \dots, y_n$ that are sampled independently from a two-part mixture of Normals model with density

$$f(y \mid \lambda) = \lambda\varphi(y \mid \mu_1, \sigma_1^2) + (1 - \lambda)\varphi(y \mid \mu_2, \sigma_2^2).$$

We can simulate some data from this model.

```r
mu1 <- 1
s1 <- 2
mu2 <- 4
s2 <- 1
lambda0 <- 0.4
n <- 100
set.seed(2017-09-12)
z <- rbinom(n, 1, lambda0)     ## "Missing" data
x <- rnorm(n, mu1 * z + mu2 * (1-z), s1 * z + (1-z) * s2)
hist(x)
rug(x)
```



For the purposes of this example, let's assume that μ_1, μ_2, σ_1^2, and σ_2^2 are known. The only unknown parameter is λ, the mixing proportion. The observed data log-likelihood is

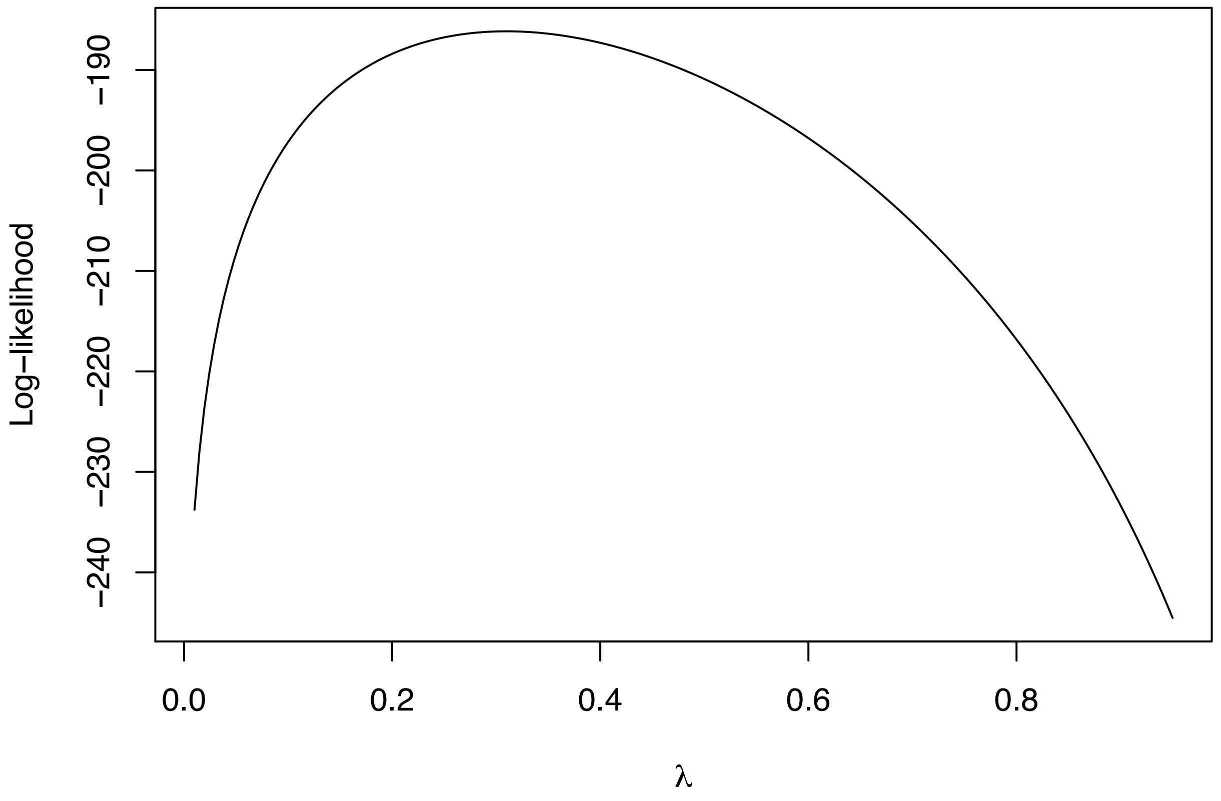
$$\log f(y_1, \dots, y_n \mid \lambda) = \log \sum_{i=1}^{n} \lambda\varphi(y_i \mid \mu_1, \sigma_1^2) + (1 - \lambda)\varphi(y_i \mid \mu_2, \sigma_2^2).$$

We can plot the observed data log-likelihood in this case with the simulated data above. First, we can write a function encoding the mixture density as a function of the data and λ.

```r
f <- function(x, lambda) {
        lambda * dnorm(x, mu1, s1) + (1-lambda) * dnorm(x, mu2, s2)
}
```

Then we can write the log-likelihood as a function of λ and plot it.

```r
loglike <- function(lambda) {
        sum(log(f(x, lambda)))
}
loglike <- Vectorize(loglike, "lambda")  ## Vectorize for plotting
par(mar = c(5,4, 1, 1))
curve(loglike, 0.01, 0.95, n = 200, ylab = "Log-likelihood",
      xlab = expression(lambda))
```



Note that the true value is $\lambda = 0.4$. We can compute the maximum likelihood estimate in this simple case with

```r
op <- optimize(loglike, c(0.1, 0.9), maximum = TRUE)
op$maximum
```

```
[1] 0.3097435
```

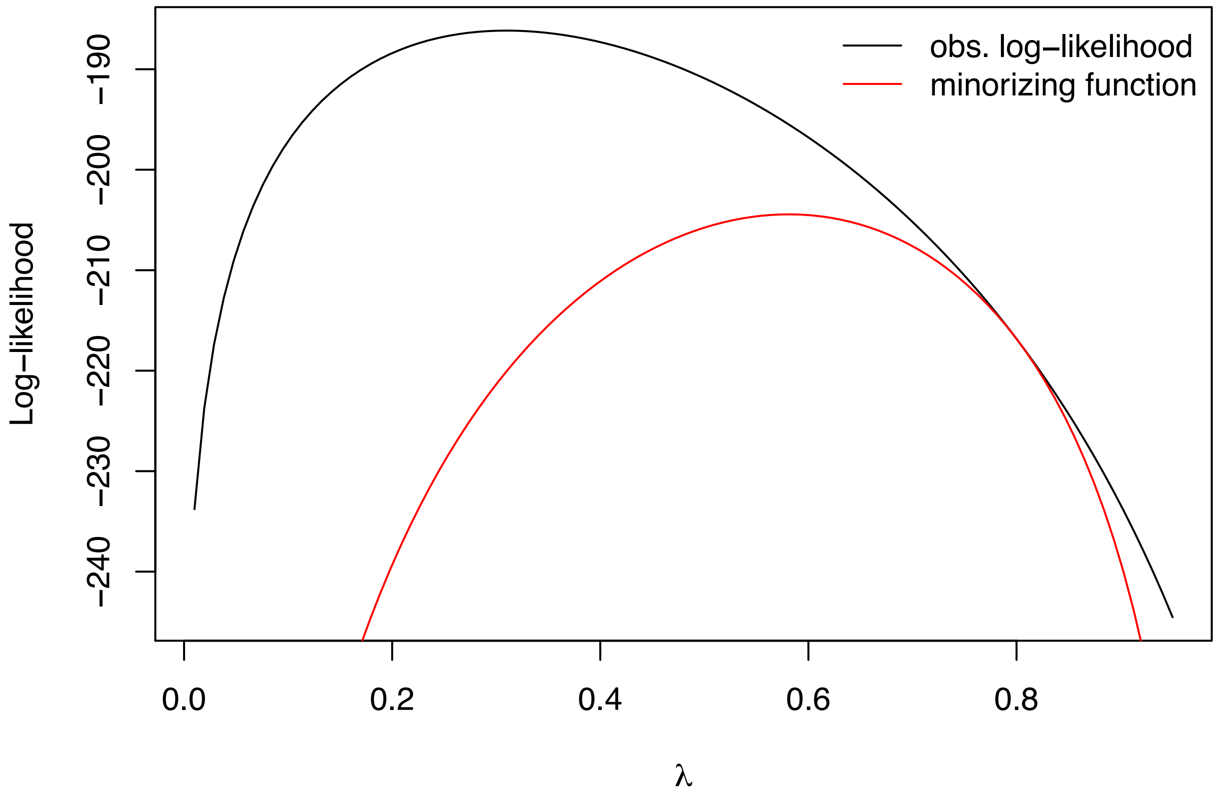
In this case it would appear that the maximum likelihood estimate exhibits some bias, but we won't worry about that right now.

We can illustrate how the minorizing function works by starting with an initial value of $\lambda_0 = 0.8$.

```r
lam0 <- 0.8
minor <- function(lambda) {
        p1 <- sum(log(f(x, lam0)))
        pi <- lam0 * dnorm(x, mu1, s1) / (lam0 * dnorm(x, mu1, s1)
                                          + (1 - lam0) * dnorm(x, mu2, s2))
        p2 <- sum(pi * dnorm(x, mu1, s1, log = TRUE)
                  + (1-pi) * dnorm(x, mu2, s2, log = TRUE)
                  + pi * log(lambda)
                  + (1-pi) * log(1-lambda))
        p3 <- sum(pi * dnorm(x, mu1, s1, log = TRUE)
                  + (1-pi) * dnorm(x, mu2, s2, log = TRUE)
                  + pi * log(lam0)
                  + (1-pi) * log(1-lam0))
        p1 + p2 - p3
}
minor <- Vectorize(minor, "lambda")
```

Now we can plot the minorizing function along with the observed log-likelihood.

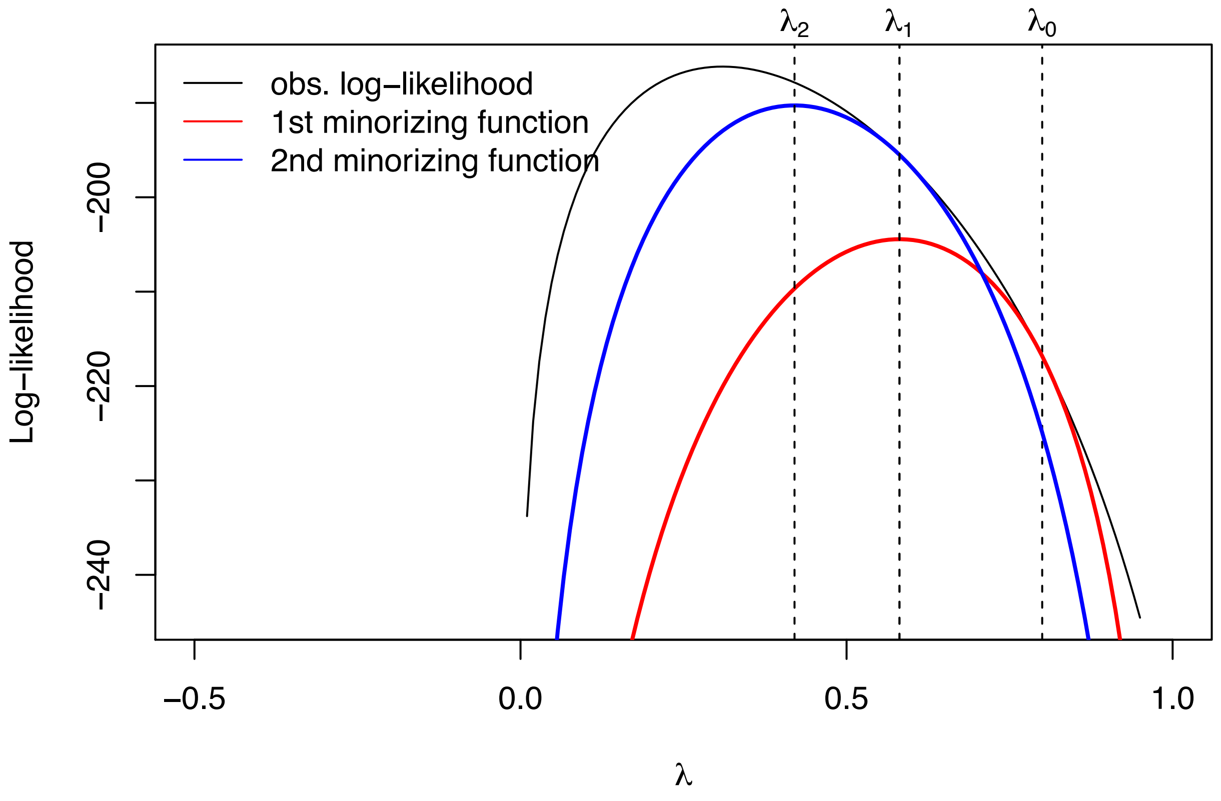
```r
par(mar = c(5,4, 1, 1))
curve(loglike, 0.01, 0.95, ylab = "Log-likelihood",
      xlab = expression(lambda))
curve(minor, 0.01, 0.95, add = TRUE, col = "red")
legend("topright", c("obs. log-likelihood", "minorizing function"),
       col = 1:2, lty = 1, bty = "n")
```



Maximizing the minorizing function gives us the next estimate of λ in the EM algorithm. It's clear from the picture that maximizing the minorizing function will increase the observed log-likelihood.

```r
par(mar = c(5,4, 2, 1))
curve(loglike, 0.01, 0.95, ylab = "Log-likelihood",
      xlab = expression(lambda), xlim = c(-0.5, 1),
      ylim = c())
abline(v = lam0, lty = 2)
mtext(expression(lambda[0]), at = lam0, side = 3)
```

```r
curve(minor, 0.01, 0.95, add = TRUE, col = "red", lwd = 2)
op <- optimize(minor, c(0.1, 0.9), maximum = TRUE)
abline(v = op$maximum, lty = 2)
lam0 <- op$maximum
curve(minor, 0.01, 0.95, add = TRUE, col = "blue", lwd = 2)
abline(v = lam0, lty = 2)
mtext(expression(lambda[1]), at = lam0, side = 3)
op <- optimize(minor, c(0.1, 0.9), maximum = TRUE)
abline(v = op$maximum, lty = 2)
mtext(expression(lambda[2]), at = op$maximum, side = 3)
legend("topleft",
       c("obs. log-likelihood", "1st minorizing function", "2nd minorizing function"),
       col = c(1, 2, 4), lty = 1, bty = "n")
```



In the figure above, the second minorizing function is constructed using λ_1 and maximized to get λ_2. This process of constructing the minorizing function and maximizing can be repeated until convergence. This is the EM algorithm at work!

4.3.2 Constrained Minimization With and Adaptive Barrier

The flip side of minorization is majorization, which is used in minimization problems. We can implement a constrained minimization procedure by creating a surrogate function that majorizes the target function and satisfies the constraints. Specifically, the goal is to minimize a funtion $f(\theta)$ subject to a set of constraints of the form $g_i(\theta) \geq 0$ where

$$g_i(\theta) = u_i'\theta - c_i$$

and where u_i is a vector of the same length as θ, c_i is a constant, and $i = 1, \dots, \ell$. These constraints are *linear* constraints on the parameters. Given the constraints and θ_n, the estimate of θ at iteration n, we can construct the surrogate function,

$$R(\theta \mid \theta_n) = f(\theta) - \lambda \sum_{i=1}^{\ell} g_i(\theta_n) \log g_i(\theta) - u_i'\theta$$

with $\lambda > 0$.

4.4 Missing Information Principle

So far, we have described the EM algorithm for computing maximum likelihood estimates in some missing data problems. But the original presentation of the EM algorithm did not discuss how to obtain any measures of uncertainty, such as standard errors. One obvious candidate would be the *observed information matrix*. However, much like with the observed log-likelihood, the observed information matrix is difficult to compute because of the missing data.

Recalling the notation from the previous section, let $f(y \mid \theta)$ be the observed data density, $g(y, z \mid \theta)$ the complete data density, and $h(z \mid y, \theta) := g(y, z \mid \theta)/f(y \mid \theta)$ the missing data density. From this we can write the following series of identities:

$$\begin{aligned}
f(y \mid \theta) &= \frac{g(y, z \mid \theta)}{h(z \mid y, \theta)} \\
-\log f(y \mid \theta) &= -\log g(y, z \mid \theta) - [-\log h(z \mid y, \theta)] \\
\mathbb{E}\left[-\frac{\partial}{\partial\theta\partial\theta'} \log f(y \mid \theta)\right] &= \mathbb{E}\left[-\frac{\partial}{\partial\theta\partial\theta'} \log g(y, z \mid \theta)\right] - \mathbb{E}\left[-\frac{\partial}{\partial\theta\partial\theta'} \log h(z \mid y, \theta)\right] \\
I_Y(\theta) &= I_{Y,Z}(\theta) - I_{Z|Y}(\theta)
\end{aligned}$$

Here, we refer to $I_Y(\theta)$ as the observed data information matrix, $I_{Y,Z}(\theta)$ as the complete data information matrix, and $I_{Z|Y}(\theta)$ as the missing information matrix. This identity allows for the the nice interpretation as the "observed information" equals the "complete information" minus the "missing information".

If we could easily evaluate the $I_Y(\theta)$, we could simply plug in the maximum likelihood estimate $\hat{\theta}$ and obtain standard errors from $I_Y(\hat{\theta})$. However, beause of the missing data, $I_Y(\theta)$ is difficult ot evaluate. Presumably, $I_{Y,Z}(\theta)$ is reasonable to compute because it is based on the complete data. What then is $I_{Z|Y}(\theta)$, the missing information matrix?

Let $S(y \mid \theta) = \frac{\partial}{\partial\theta} \log f(y \mid \theta)$ be the observed score function and let $S(y, z \mid \theta) = \frac{\partial}{\partial\theta} \log g(y, z \mid \theta)$ be the complete data score function. In a critically important paper, Tom Louis showed that

$$I_{Z|Y}(\theta) = \mathbb{E}\left[S(y, z \mid \theta)S(y, z \mid \theta)'\right] - S(y \mid \theta)S(y \mid \theta)'.$$

with the expectation taken with respect to the missing data density $h(z \mid y, \theta)$. The first part of the right hand side involves computations on the complete data, which is fine. Unfortunately, the second part involves the observed score function, which is presumably difficult to evaluate. However, by definition, $S(y \mid \hat{\theta}) = 0$ at the maximum likelihood estimate $\hat{\theta}$. Therefore, we can write the observed information matrix at the MLE as

$$I_Y(\hat{\theta}) = I_{Y,Z}(\hat{\theta}) - \mathbb{E}\left[S(y, z \mid \theta)S(y, z \mid \theta)'\right]$$

so that all computations are done on the complete data. Note also that

$$\begin{aligned}
I_{Y,Z}(\hat{\theta}) &= -\mathbb{E}\left[\frac{\partial}{\partial\theta\partial\theta'} \log g(y, z \mid \theta) \middle| \hat{\theta}, y\right] \\
&= -Q''(\hat{\theta} \mid \hat{\theta})
\end{aligned}$$

Meilijson showed that when the observed data $\mathbf{y} = y_1, \dots, y_n$ are iid, then

$$S(\mathbf{y} \mid \theta) = \sum_{i=1}^{n} S(y_i \mid \theta)$$

and hence

$$\begin{aligned} I_Y(\theta) &= \operatorname{Var}(S(\mathbf{y} \mid \theta)) \\ &= \frac{1}{n}\sum_{i=1}^{n} S(y_i \mid \theta)S(y_i \mid \theta)' - \frac{1}{n^2}S(\mathbf{y} \mid \theta)S(\mathbf{y} \mid \theta)'. \end{aligned}$$

Again, because $S(\mathbf{y} \mid \hat{\theta}) = 0$ at the MLE, we can ignore the second part of the expression if we are interested in obtaining the observed information at the location of the MLE. As for the first part of the expression, Louis also showed that

$$S(y_i \mid \theta) = \mathbb{E}[S(y_i, z_i \mid \theta) \mid y_i, \theta_0].$$

where the expectation is once again taken with respect to the missing data density. Therefore, we can transfer computations on the observed score function to computations on the complete score function.

4.5 Acceleration Methods

Dempster et al. showed that the convergence rate for the EM algorithm is linear, which can be painfully slow for some problems. Therefore, a cottage industry has developed around the notion of speeding up the convergence of the algorithm. Two approaches that we describe here are one proposed by Tom Louis based on the Aitken acceleration technique and the SQUAREM approach of Varadhan and Roland.

4.5.1 Louis's Acceleration

If we let $M(\theta)$ be the map representing a single iteration of the EM algorithm, so that $\theta_{n+1} = M(\theta_n)$. Then under standard regularity conditions, we can approximate M near the optimum value $\theta^\star$ with

$$\theta_{n+1} = M(\theta_n) \approx \theta_n + J(\theta_{n-1})(\theta_n - \theta_{n-1})$$

where Dempster et al. 1977 showed that J is

$$J(\theta) = I_{Z|Y}(\theta)I_{Z,Y}(\theta)^{-1},$$

which can be interpreted as characterizing the proportion of missing data. (Dempster et al. also showed that the rate of convergence of the EM algorithm is determined by the modulus of the largest eigenvalue of $J(\theta^\star)$.)

Furthermore, for large j and n, we have

$$\theta_{n+j+1} - \theta_{n+j} \approx J^{(n)}(\theta^\star)(\theta_{j+1} - \theta_j)$$

where $\theta^\star$ is the MLE, and $J^{(n)}(\theta^\star)$ is J multiplied by itself n times. Then if $\theta^\star$ is the limit of the sequence $\{\theta_n\}$, we can write (trivially) for any j

$$\theta^\star = \theta_j + \sum_{k=1}^{\infty}(\theta_{k+j} - \theta_{k+j-1})$$

We can then approximate this with

$$\begin{aligned}\theta^\star &\approx \theta_j + \left(\sum_{k=0}^{\infty} J^{(k)}(\theta^\star)\right)(\theta_{j+1}-\theta_j)\\ &= \theta_j + (I - J(\theta^\star))^{-1}(\theta_{j+1}-\theta_j)\end{aligned}$$

The last equivalence is possible because the eigenvalues of J are all less than one in absolute value.

Given this relation, the acceleration method proposed by Louis works as follows. Given θ_n, the current estimate of θ,

1. Compute θ_{n+1} using the standard EM algorithm
2. Compute $(I-\hat{J})^{-1} = I_{Y,Z}(\theta_n)I_Y(\theta_n)^{-1}$
3. Let $\theta^\star = \theta_n + (I-\hat{J})^{-1}(\theta_{n+1}-\theta_n)$.
4. Set $\theta_{n+1} = \theta^\star$.

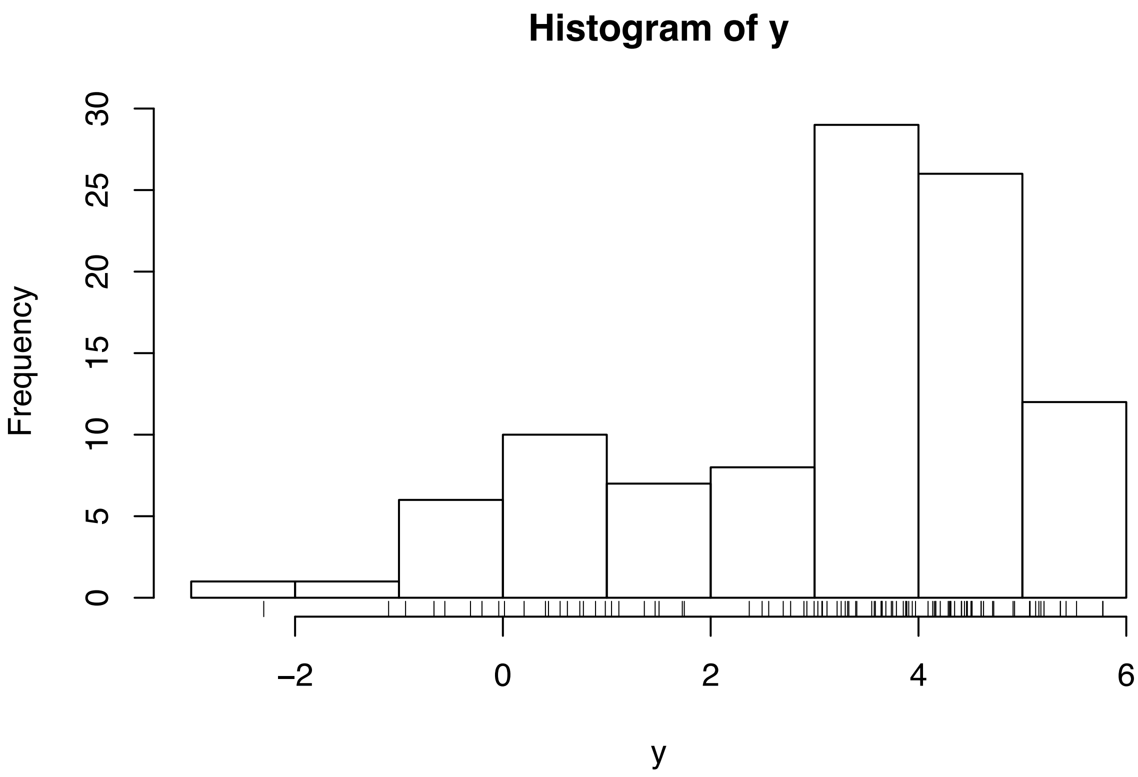
The cost of using Louis's technique is minimal if the dimension of θ is small. Ultimately, it comes down to the cost of inverting $I_Y(\theta_n)$ relative to running a single iteration of the EM algorithm. Further, it's worth emphasizing that the convergence of the approach is only guaranteed for values of θ in a neighborhood of the optimum $\theta^\star$, but the size and nature of that neighborhood is typically unknown in applications.

Looking at the algorithm described above, we can gather some basic heuristics of how it works. When the information in the observed data is high relative to the complete data, then the value of $(I-\hat{J})^{-1}$ will be close to 1 and the sequence of iterates generated by the algorithm will be very similar to the usual EM sequence. However, if the proportion of missing data is high, then $(I-\hat{J})^{-1}$ will be much greater than 1 and the modifications that the algorithm makes to the usual EM sequence will be large.

4.5.1.1 Example: Normal Mixture Model

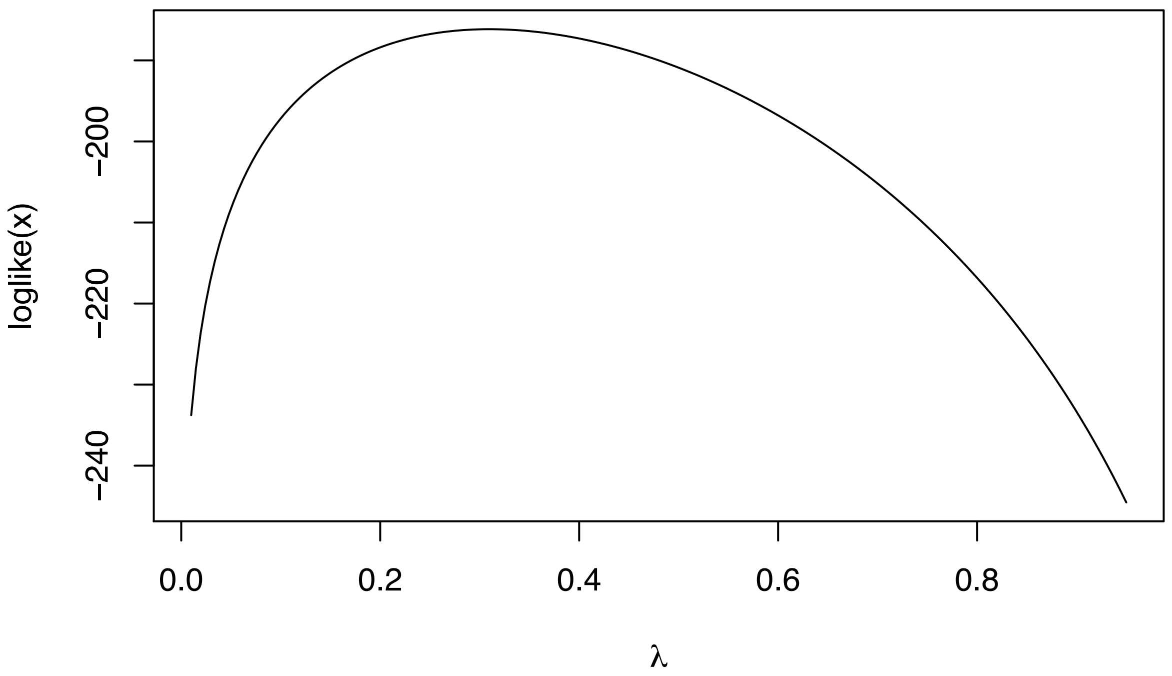
Recall that the data are generated as follows.

```r
mu1 <- 1
s1 <- 2
mu2 <- 4
s2 <- 1
lambda0 <- 0.4
n <- 100
set.seed(2017-09-12)
z <- rbinom(n, 1, lambda0)
y <- rnorm(n, mu1 * z + mu2 * (1-z), s1 * z + (1-z) * s2)
hist(y)
rug(y)
```



If we assume μ_1, μ_2, σ_1 and σ_2 are known, then we can visualize the observed data log-likelihood as a function of λ.

```r
f <- function(y, lambda) {
        lambda * dnorm(y, mu1, s1) + (1-lambda) * dnorm(y, mu2, s2)
}
loglike <- Vectorize(
        function(lambda) {
                sum(log(f(y, lambda)))
        }
)
curve(loglike, 0.01, 0.95, n = 200, xlab = expression(lambda))
```



Because the observed log-likelihood is relatively simple in this case, we can maximize it directly and obtain the true maximum likelihood estimate.

```
op <- optimize(loglike, c(0.01, 0.95), maximum = TRUE, tol = 1e-8)
op$maximum
```

```
[1] 0.3097386
```

We can encode the usual EM iteration as follows. The `M` function represents a single iteration of the EM algorithm as a function of the current value of λ.

```
make_pi <- function(lambda, y, mu1, mu2, s1, s2) {
        lambda * dnorm(y, mu1, s1) / (lambda * dnorm(y, mu1, s1) +
                                      (1 - lambda) * (dnorm(y, mu2, s2)))
}
M <- function(lambda0) {
        pi.est <- make_pi(lambda0, y, mu1, mu2, s1, s2)
        mean(pi.est)
}
```

We can also encode the accelerated version here with the function `Mstar`. The functions `Iy` and `Iyz` encode the observed and complete data information matrices.

```
Iy <- local({
        d <- deriv3(~ log(lambda * dnorm(y, mu1, s1) + (1-lambda) * dnorm(y, mu2, s2)),
                    "lambda", function.arg = TRUE)
        function(lambda) {
                H <- attr(d(lambda), "hessian")
                sum(H)
        }
})

Iyz <- local({
        d <- deriv3(~ pihat * log(lambda) + (1-pihat) * log(1-lambda),
                    "lambda", function.arg = TRUE)
        function(lambda) {
                H <- attr(d(lambda), "hessian")
                sum(H)
        }
})

Mstar <- function(lambda0) {
        lambda1 <- M(lambda0)
        pihat <- make_pi(lambda0, y, mu1, mu2, s1, s2)
        lambda0 + (Iyz(lambda0) / Iy(lambda0)) * (lambda1 - lambda0)
}
```

Taking a starting value of $\lambda = 0.1$, we can see the speed at which the original EM algorithm and the accelerated versions converge toward the MLE.

```
lambda0 <- 0.1
lambda0star <- 0.1
iter <- 6
EM <-  numeric(iter)
Accel <- numeric(iter)
for(i in 1:iter) {
        pihat <- make_pi(lambda0, y, mu1, mu2, s1, s2)
        lambda1 <- M(lambda0)
        lambda1star <- Mstar(lambda0star)
        EM[i] <- lambda1
        Accel[i] <- lambda1star
        lambda0 <- lambda1
        lambda0star <- lambda1star
}
results <- data.frame(EM = EM, Accel = Accel,
                      errorEM = abs(EM - op$maximum),
                      errorAccel = abs(Accel - op$maximum))
```

After six iterations, we have the following.

```
format(results, scientific = FALSE)
```

```
         EM     Accel     errorEM        errorAccel
1 0.2354541 0.2703539 0.0742845130 0.039384732683244
2 0.2850198 0.3075326 0.0247188026 0.002206035238572
3 0.3014516 0.3097049 0.0082869721 0.000033703087859
4 0.3069478 0.3097384 0.0027907907 0.000000173380975
5 0.3087971 0.3097386 0.0009414640 0.000000006066448
6 0.3094208 0.3097386 0.0003177924 0.000000005785778
```

One can see from the `errorAccel` column that the accelerated method's error decreases much faster than the standard method's error (in the `errorEM` column). The accelerated method appears to be close to the MLE by the third iteration whereas the standard EM algorithm hasn't quite gotten there by six iterations.

4.5.2 SQUAREM

Let $M(\theta)$ be the map representing a single iteration of the EM algorithm so that $\theta_{n+1} = M(\theta_n)$. Given the current value θ_0,

1. Let $\theta_1 = M(\theta_0)$

2. Let $\theta_2 = M(\theta_1)$
3. Compute the difference $r = \theta_1 - \theta_0$
4. Let $v = (\theta_2 - \theta_1) - r$
5. Compute the step length α
6. Modify α if necessary
7. Let $\theta' = \theta_0 - 2\alpha r + \alpha^2 v$
8. Let $\theta_1 = M(\theta')$
9. Compare θ_1 with θ_0 and check for convergence. If we have not yet converged, let $\theta_0 = \theta_1$ and go back to Step 1.

5 Integration

In statistical applications we often need to compute quantities of the form

$$\mathbb{E}_f g(X) = \int g(x)f(x)\,dx$$

where X is a random variable drawn from a distribution with probability mass function f. Another quantity that we often need to compute is the normalizing constant for a probability density function. If X has a density that is proportional to $p(x \mid \theta)$ then its normalizing constant is $\int p(x \mid \theta)\,dx$.

In both problems—computing the expectation and computing the normalizing constant—an integral must be evaluated.

Approaches to solving the integration problem roughly fall into two categories. The first categories involves identifying a sequence of estimates to the integral that eventually converge to the true value as some index (usually involving time or resources) goes to infinity. Adaptive quadrature, independent Monte Carlo and Markov chain Monte Carlo techniques all fall into this category. Given enough time and resources, these techniques should converge to the true value of the integral.

The second category of techniques involves identifying a class of alternative functions that are easier to work with, finding the member of that class that best matches the true function, and then working with the alternate function instead to compute the integral. Laplace approximation, variational inference, and approximate Bayes computation (ABC) fall into this category of approaches. For a given dataset, these approaches will not provide the true integral value regardless of time and resources, but as the sample size increases, the approximations will get better.

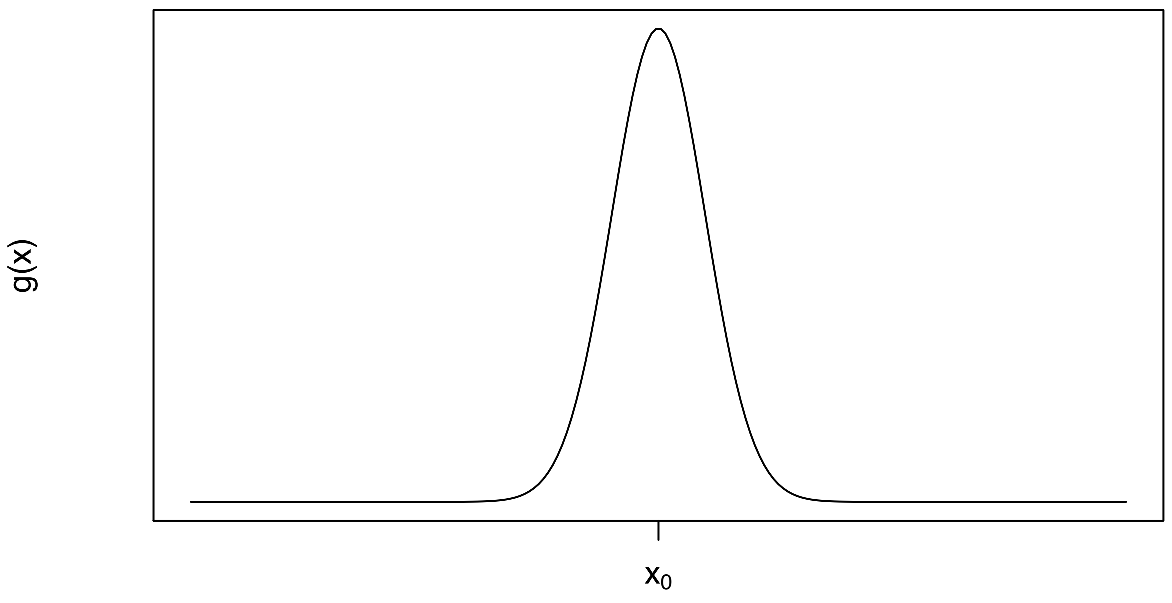
5.1 Laplace Approximation

The first technique that we will discuss is Laplace approximation. This technique can be used for reasonably well behaved functions that have most of their mass concentrated in a small area of their domain. Technically, it works for functions that are in the class of $\mathfrak{L}^2$, meaning that

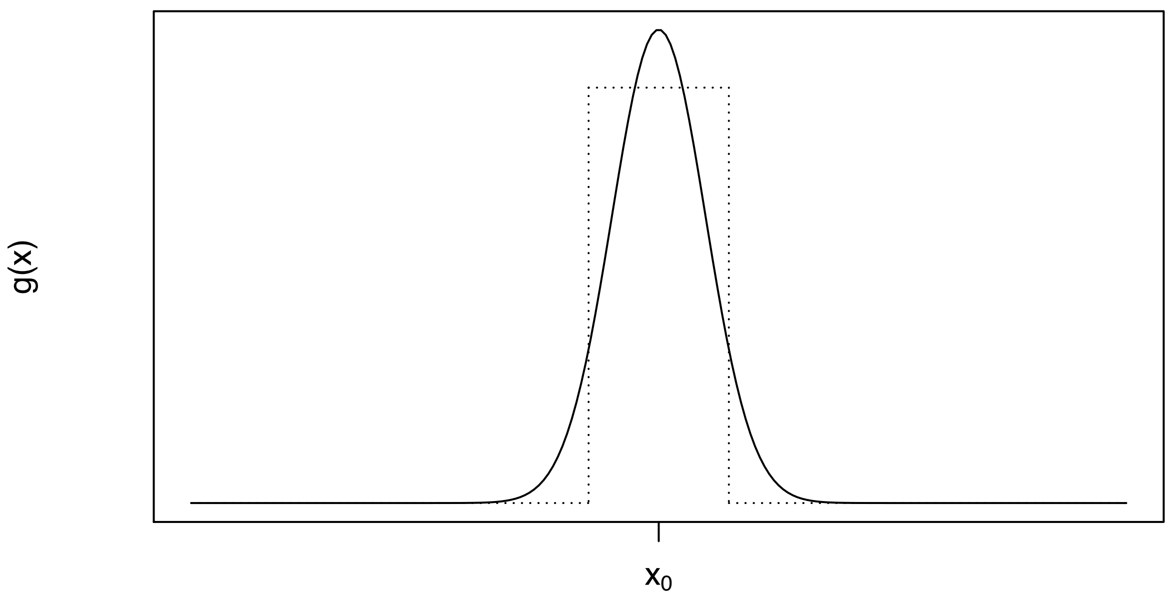
$$\int g(x)^2\,dx < \infty$$

Such a function generally has very rapidly decreasing tails so that in the far reaches of the domain we would not expect to see large spikes.

Imagine a function that looks as follows



We can see that this function has most of its mass concentrated around the point x_0 and that we could probably approximate the area under the function with something like a step function.



The benefit of using something like a step function is that the area under a step function is trivial to compute. If we could find a principled and automatic way to find that approximating step function, and it were easier than just directly computing the integral in the first place, then we could have an alternative to computing the integral. In other words, we could perhaps say that

$$\int g(x)\,dx \approx g(x_0)\varepsilon$$

for some small value of ε.

In reality, we actually have some more sophisticated functions that we can use besides step functions, and that's how the Laplace approximation works. **The general idea is to take a well-behaved uni-modal function and approximate it with a Normal density function**, which is a very well-understood quantity.

Suppose we have a function $g(x) \in \mathfrak{L}^2$ which achieves its maximum at x_0. We want to compute

$$\int_a^b g(x)\, dx.$$

Let $h(x) = \log g(x)$ so that we have

$$\int_a^b g(x)\, dx = \int_a^b \exp(h(x))\, dx$$

From here we can take a Taylor series approximation of $h(x)$ around the point x_0 to give us

$$\int_a^b \exp(h(x))\, dx \approx \int_a^b \exp\left(h(x_0) + h'(x_0)(x - x_0) + \frac{1}{2}h''(x_0)(x - x_0)^2\right)\, dx$$

Because we assumed $h(x)$ achieves its maximum at x_0, we know $h'(x_0) = 0$. Therefore, we can simplify the above expression to be

$$= \int_a^b \exp\left(h(x_0) + \frac{1}{2}h''(x_0)(x - x_0)^2\right)\, dx$$

Given that $h(x_0)$ is a constant that doesn't depend on x, we can pull it outside the integral. In addition, we can rearrange some of the terms to give us

$$= \exp(h(x_0)) \int_a^b \exp\left(-\frac{1}{2}\frac{(x - x_0)^2}{-h''(x_0)^{-1}}\right)\, dx$$

Now that looks more like it, right? Inside the integral we have a quantity that is proportional to a Normal density with mean x_0 and variance $-h''(x_0)^{-1}$. At this point we are just one call to the `pnorm()` function away from approximating our integral. All we need is to compute our normalizing constants.

If we let $\Phi(x \mid \mu, \sigma^2)$ be the cumulative distribution function for the Normal distribution with mean μ and variance σ^2 (and φ is its density function), then we can write the above expression as

$$\begin{aligned}
&= \exp(h(x_0))\sqrt{\frac{2\pi}{-h''(x_0)}} \int_a^b \varphi(x \mid x_0, -h''(x_0)^{-1})\, dx \\
&= \exp(h(x_0))\sqrt{\frac{2\pi}{-h''(x_0)}} \left[\Phi\left(b \mid x_0, -h''(x_0)^{-1}\right) - \Phi\left(a \mid x_0, -h''(x_0)^{-1}\right)\right]
\end{aligned}$$

Recall that $\exp(h(x_0)) = g(x_0)$. If $b = \infty$ and $a = -\infty$, as is commonly the case, then the term in the square brackets is equal to 1, making the Laplace approximation equal to the value of the function $g(x)$ at its mode multiplied by a constant that depends on the curvature of the function h.

One final note about the Laplace approximation is that it replaces the problem of integrating a function with the problem of *maximizing* it. In order to compute the Laplace approximation, we have to compute the location of the mode, which is an optimization problem. Often, this problem is faster to solve using well-understood function optimizers than integrating the same function would be.

5.1.1 Computing the Posterior Mean

In Bayesian computations we often want to compute the posterior mean of a parameter given the observed data. If y represents data we observe and y comes from the distribution $f(y \mid \theta)$ with parameter θ and θ has a prior distribution $\pi(\theta)$, then we usually want to compute the posterior distribution $p(\theta \mid y)$ and its mean,

$$\mathbb{E}_p[\theta] = \int \theta \, p(\theta \mid y) \, d\theta.$$

We can then write

$$\begin{aligned}\int \theta \, p(\theta \mid y) \, dx &= \frac{\int \theta \, f(y \mid \theta)\pi(\theta) \, d\theta}{\int f(y \mid \theta)\pi(\theta) \, d\theta} \\ &= \frac{\int \theta \, \exp(\log f(y \mid \theta)\pi(\theta)) \, d\theta}{\int \exp(\log f(y \mid \theta)\pi(\theta) \, d\theta)}\end{aligned}$$

Here, we've used the age old trick of exponentiating and log-ging.

If we let $h(\theta) = \log f(y \mid \theta)\pi(\theta)$, then we can use the same Laplace approximation procedure described in the previous section. However, in order to do that we must know where $h(\theta)$ achieves its maximum. Because $h(\theta)$ is simply a monotonic transformation of a function proportional to the posterior density, we know that $h(\theta)$ achieves its maximum at the *posterior mode.*

Let $\hat{\theta}$ be the posterior mode of $p(\theta \mid y)$. Then we have

$$\begin{aligned}\int \theta \, p(\theta \mid y) \, dx &\approx \frac{\int \theta \exp\left(h(\hat{\theta}) + \frac{1}{2}h''(\hat{\theta})(\theta - \hat{\theta})^2\right) \, d\theta}{\int \exp\left(h(\hat{\theta}) + \frac{1}{2}h''(\hat{\theta})(\theta - \hat{\theta})^2\right) \, d\theta} \\ &= \frac{\int \theta \exp\left(\frac{1}{2}h''(\hat{\theta})(\theta - \hat{\theta})^2\right) \, d\theta}{\int \exp\left(\frac{1}{2}h''(\hat{\theta})(\theta - \hat{\theta})^2\right) \, d\theta} \\ &= \frac{\int \theta \sqrt{\frac{2\pi}{-h''(\hat{\theta})}} \varphi\left(\theta \mid \hat{\theta}, -h''(\hat{\theta})^{-1}\right) \, d\theta}{\int \sqrt{\frac{2\pi}{-h''(\hat{\theta})}} \varphi\left(\theta \mid \hat{\theta}, -h''(\hat{\theta})^{-1}\right) \, d\theta} \\ &= \hat{\theta}\end{aligned}$$

Hence, the Laplace approximation to the posterior mean is equal to the posterior mode. This approximation is likely to work well when the posterior is unimodal and relatively symmetric around the model. Furthermore, the more concentrated the posterior is around $\hat{\theta}$, the better.

5.1.1.1 Example: Poisson Data with a Gamma Prior

In this simple example, we will use data drawn from a Poisson distribution with a mean that has a Gamma prior distribution. The model is therefore

$$\begin{aligned}Y \mid \mu &\sim \text{Poisson}(\mu) \\ \mu &\sim \text{Gamma}(a, b)\end{aligned}$$

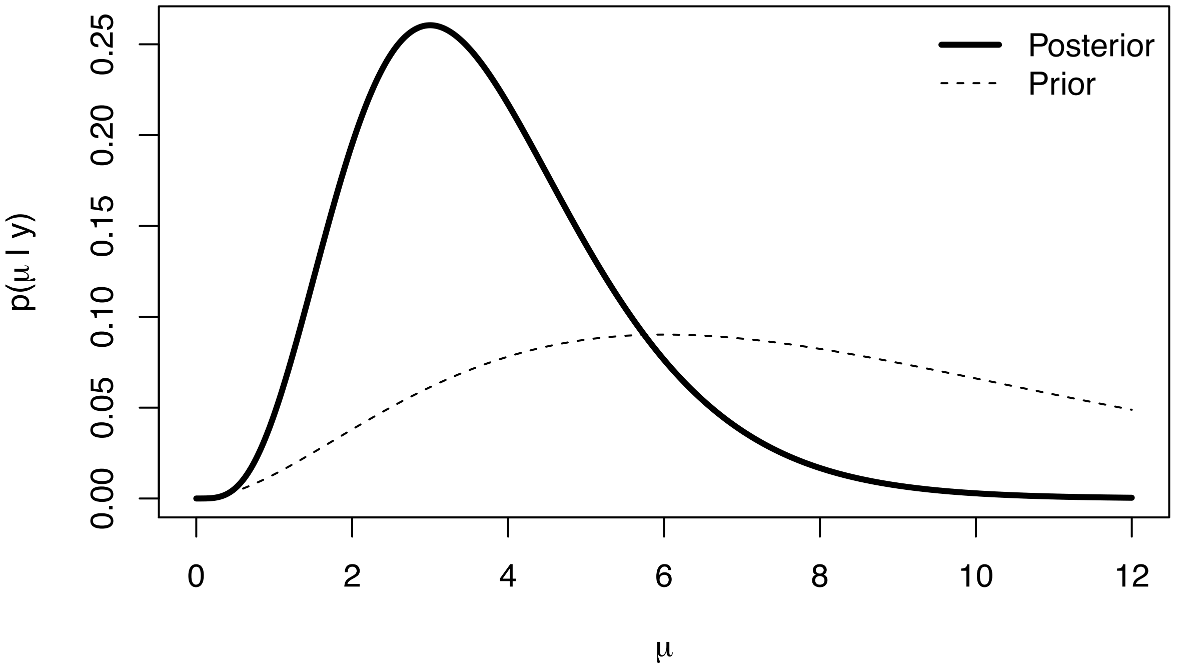
where the Gamma density is

$$f(\mu) = \frac{1}{b^a\Gamma(a)}\mu^{a-1}e^{-\mu/b}$$

In this case, given an observation y, the posterior distribution is simply a Gamma distribution with shape parameter $y + a$ and scale parameter $(1 + 1/b)$.

Suppose we observe $y = 2$. We can draw the posterior distribution and prior distribution as follows.

```
make_post <- function(y, shape, scale) {
        function(x) {
                dgamma(x, shape = y + shape,
                       scale = 1 / (1 + 1 / scale))
        }
}
set.seed(2017-11-29)
y <- 2
prior.shape <- 3
prior.scale <- 3
p <- make_post(y, prior.shape, prior.scale)
curve(p, 0, 12, n = 1000, lwd = 3, xlab = expression(mu),
      ylab = expression(paste("p(", mu, " | y)")))
curve(dgamma(x, shape = prior.shape, scale = prior.scale), add = TRUE,
      lty = 2)
legend("topright", legend = c("Posterior", "Prior"), lty = c(1, 2), lwd = c(3, 1), bty = "n")
```



Because this is a Gamma distribution, we can also compute the posterior mode in closed form.

```
pmode <- (y + prior.shape - 1) * (1 / (1 + 1 / prior.scale))
pmode
```

```
[1] 3
```

We can also compute the mean.

```
pmean <- (y + prior.shape) * (1 / (1 + 1 / prior.scale))
pmean
```

```
[1] 3.75
```

From the skewness in the figure above, it's clear that the mean and the mode should not match.

We can now see what the Laplace approximation to the posterior looks like in this case. First, we can compute the gradient and Hessian of the Gamma density.

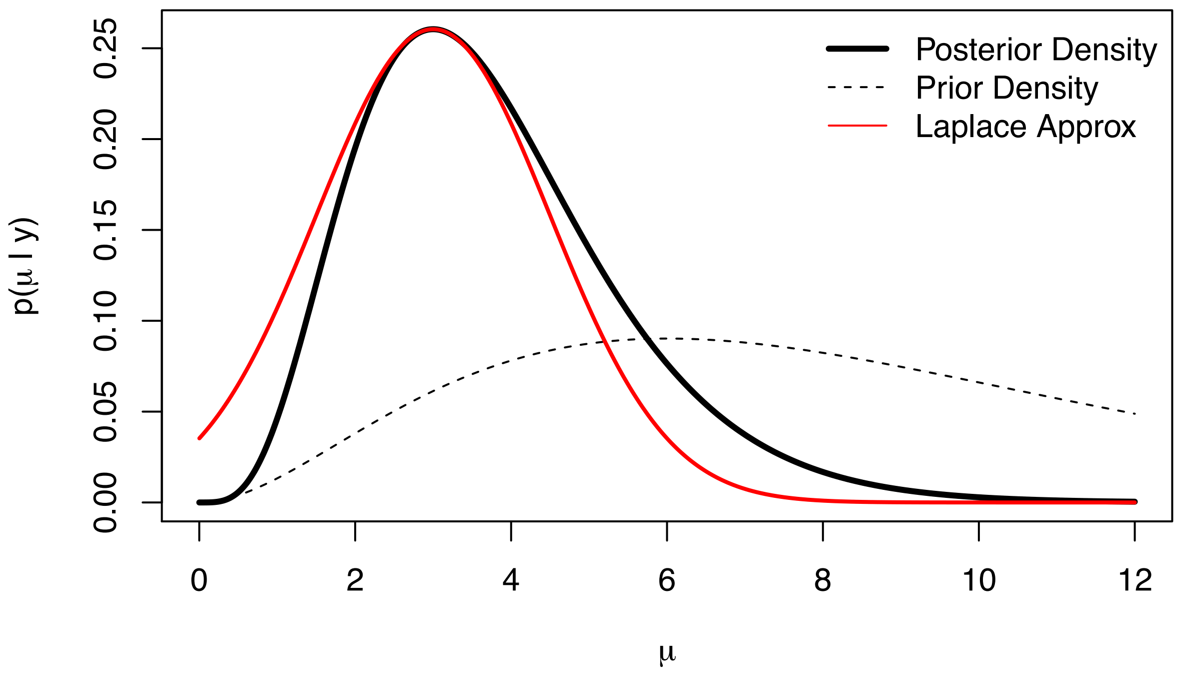
```
a <- prior.shape
b <- prior.scale
fhat <- deriv3(~ mu^(y + a - 1) * exp(-mu * (1 + 1/b)) / ((1/(1+1/b))^(y+a) * gamma(y + a)), "mu", func
```

Then we can compute the quadratic approximation to the density via the `lapprox()` function below.

```
post.shape <- y + prior.shape - 1
post.scale <- 1 / (length(y) + 1 / prior.scale)
lapprox <- Vectorize(function(mu, mu0 = pmode) {
        deriv <- fhat(mu0)
        grad <- attr(deriv, "gradient")
        hess <- drop(attr(deriv, "hessian"))
        f <- function(x) dgamma(x, shape = post.shape, scale = post.scale)
        hpp <- (hess * f(mu0) - grad^2) / f(mu0)^2
        exp(log(f(mu0)) + 0.5 * hpp * (mu - mu0)^2)
}, "mu")
```

Plotting the true posterior and the Laplace approximation gives us the following.

```
curve(p, 0, 12, n = 1000, lwd = 3, xlab = expression(mu),
      ylab = expression(paste("p(", mu, " | y)")))
curve(dgamma(x, shape = prior.shape, scale = prior.scale), add = TRUE,
      lty = 2)
legend("topright",
       legend = c("Posterior Density", "Prior Density", "Laplace Approx"),
       lty = c(1, 2, 1), lwd = c(3, 1, 1), col = c(1, 1, 2), bty = "n")
curve(lapprox, 0.001, 12, n = 1000, add = TRUE, col = 2, lwd = 2)
```



The solid red curve is the Laplace approximation and we can see that in the neighborhood of the mode, the approximation is reasonable. However, as we move farther away from the mode, the tail of the Gamma is heavier on the right.

Of course, this Laplace approximation is done with only a single observation. One would expect the approximation to improve as the sample size increases. In this case, with respect to the posterior mode as an approximation to the posterior mean, we can see that the difference between the two is simply

$$\hat{\theta}_{\text{mean}} - \hat{\theta}_{\text{mode}} = \frac{1}{n + 1/b}$$

which clearly goes to zero as $n \to \infty$.

6 Independent Monte Carlo

Suppose we want to compute for some function $h : \mathbb{R}^k \to \mathbb{R}$,

$$\mathbb{E}_f[h(X)] = \int h(x) f(x)\, dx.$$

If we could simulate $x_1, \ldots, x_n \overset{\text{i.i.d.}}{\sim} f$, then by the law of large numbers, we would have

$$\frac{1}{n} \sum_{i=1}^{n} h(x_i) \longrightarrow \mathbb{E}_f[h(X)].$$

Furthermore, we have that

$$\operatorname{Var}\left(\frac{1}{n} \sum_{i=1}^{n} h(x_i) \right) = \frac{1}{n^2} \sum_{i=1}^{n} \operatorname{Var}(h(X_i)) = \frac{1}{n} \operatorname{Var}(h(X_1)),$$

which, we will note for now, does not depend on the dimension of the random variable X_1.

This approach to computing the expectation above is known as Monte Carlo integration which takes advantage of the law of large numbers saying that averages of numbers converge to their expectation. Monte Carlo integration can be quite useful, but it takes the problem of computing the integral directly (or via approximation) and replaces it with a problem of drawing samples from an arbitrary density function f.

Before we go further, we will take a brief diversion into random number generation.

6.1 Random Number Generation

In order to use the simulation-based techniques described in this book, we will need to be able to generate sequences of random numbers. Most of the time in the software we are using, there is a function that will do this for us. For example, in R, if we want to generate uniformly distributed random numbers over a fixed interval, we can use the `runif()` function.

Nevertheless, there are two issues that are worth considering here:

1. It is useful to know a little about what is going on under the hood of these random number generators. How exactly is the sequence of numbers created?
2. Built-in functions in R are only useful for well-known or well-characterized distributions. However, wil many simulation-based techniques, we will want to generate random numbers from distributions that we have likely never seen before and for which there will not be any built-in function.

6.1.1 Pseudo-random Numbers

The truth is that R, along with most other analytics packages, does not generate genuine random numbers. R generates *pseudo-random numbers* that appear to be random but are actually generated in a deterministic way. This approach sounds worse, but it's actually better for two reasons. First, generating genuine random numbers can be slow and often will depend on some outside source of entropy/randomness. Second, genuine random numbers are not reproducible, so if you wanted to re-create some results based on simulations, you would not ever be able to do so.

Pseudo-random number generators (PRNGs have a long history that we will not cover here. One useful thing to know is that this is tricky area and it is not simple to wander in and start developing your own PRNGs. It is useful to know how the systems work, but after that it's best to leave the specifics to the experts.

The most commonly used class of PRNGs in scientific applications is the *linear congruential generator*. The basic idea behind an LCG is that we have a starting *seed*, and then from there we generate pseudo-random numbers via a recurrence relation. Most LCGs have the form

$$X_{n+1} = (aX_n + c) \bmod m$$

where a is called the *multiplier*, c is the *increment*, and m is the *modulus*. For $n = 0$, the value X_0 is the seed. Modular arithmetic is needed in order to prevent the sequence from going off to infinity. For most generators, the values $X_0, X_1, \dots$ are integers. However, we could for example generate Uniform(0, 1) variates by taking $U_n = X_n/m$.

Given the recurrence relation above and the modular arithmetic, the maximum number of distinct values that can be generated by an LCG is m, so we would need m to be very large. The hope is that if the PRNG is well-designed, the sequence should hit every number from 0 to $m-1$ before repeating. If a number is repeated before all numbers are seen, then the generator has a period in it that is shorter than the maximal period that is possible. Setting the values of a, c, and m is a tricky business and can require some experimentation. In summary, don't do this at home. As an (historical) example, the random number generator proposed by the book *Numerical Recipes* specified that $a = 1664525$, $c = 1013904223$, and $m = 2^{32}$.

Perhaps the biggest problem with using the historical LCGs for generating random numbers is that their periods are too short, even if they manage to hit the maximal period. Given the scale of simulations being conducted today, even a period of 2^{32} would likely be too short to appear sufficiently random. Most analytical software systems have since moved on to other more sophisticated generators. For example, the default in R is the Mersenne-Twister, which has a long period of $2^{19937} - 1$.

Further notes:

- The randomness of a pseudo-random sequence can be checked via statistical tests of uniformity, such as the Kolmogorov-Smirnoff test, Chi-square test, or the Marsaglia "die hard" tests.
- Many PRNGs generate sequences that look random in one dimension but do not look random when embedded into higher dimensions. It is possible for PRNGs to generate numbers that lie on a higher-dimensional hyperplane that still look random in one dimension.

6.2 Non-Uniform Random Numbers

Uniform random numbers are useful, but usually we want to generate random numbers from some non-uniform distribution. There are a few ways to do this depending on the distribution.

6.2.1 Inverse CDF Transformation

The most generic method (but not necessarily the simplest) uses the inverse of the cumulative distribution function of the distribution.

Suppose we wnat to draw samples from a distribution with density f and cumulative distribution function $F(x) = \int_{-\infty}^{x} f(t)\,dt$. Then we can do the following:

1. Draw $U \sim \text{Unif}(0, 1)$ using any suitable PRNG.
2. Let $X = F^{-1}(U)$. Then X is distributed according to f.

Of course this method requires the inversion of the CDF, which is usually not possible. However, it works well for the Exponential(λ) distribution. Here, we have that

$$f(x) = \frac{1}{\lambda} e^{-x/lambda}$$

and

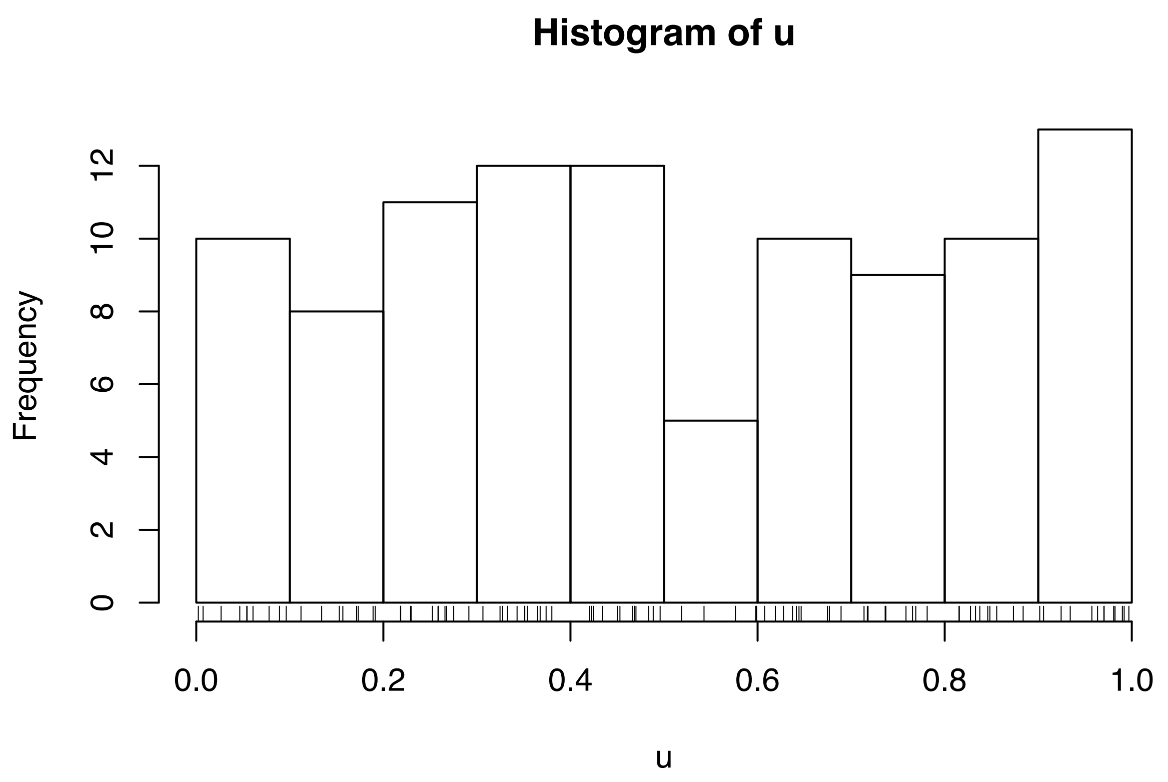
$$F(x) = 1 - e^{-x/\lambda}.$$

Therefore, the inverse of the CDF is

$$F^{-1}(u) = -\lambda \log(1 - u).$$

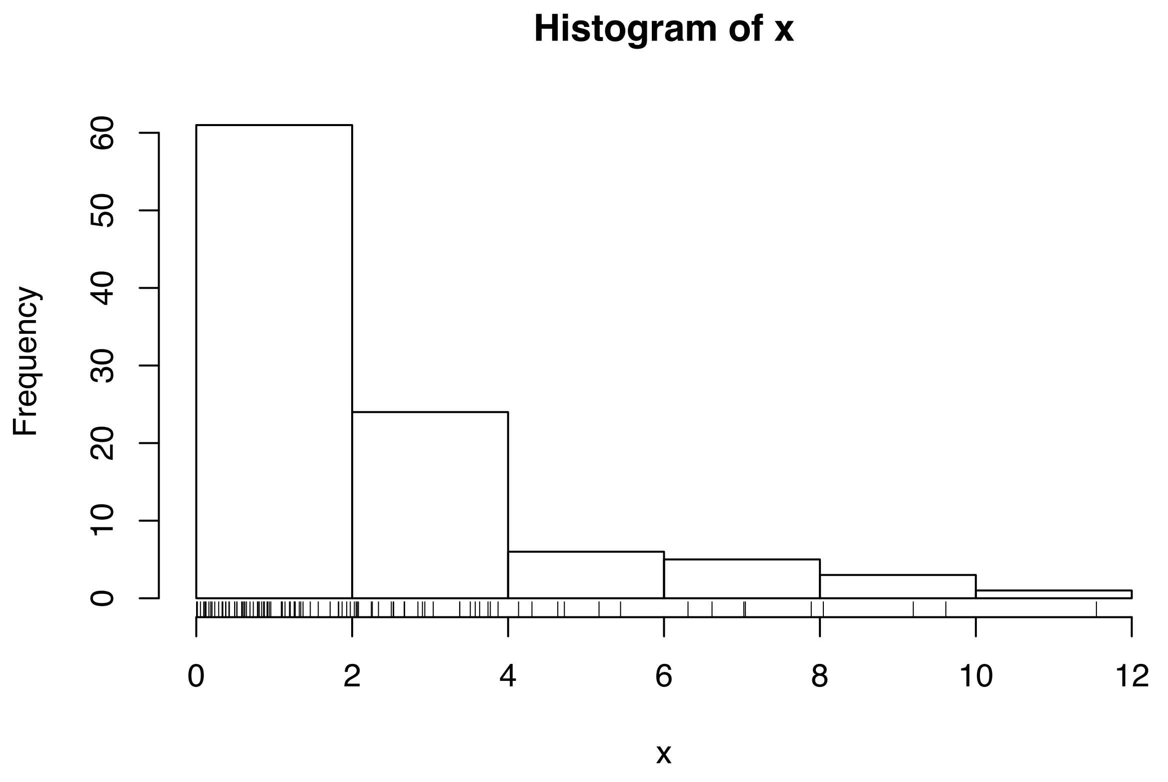
First we can draw our uniform random variables.

```
set.seed(2017-12-4)
u <- runif(100)
hist(u)
rug(u)
```



Then we can apply the inverse CDF.

```r
lambda <- 2  ## Exponential with mean 2
x <- -lambda * log(1 - u)
hist(x)
rug(x)
```



The problem with this method is that inverting the CDF is usually a difficult process and so other methods will be needed to generate other random variables.

6.2.2 Other Transformations

The inverse of the CDF is not the only function that we can use to transform uniform random variables into random variables with other distributions. Here are some common transformations.

To generate **Normal** random variables, we can

1. Generate $U_1, U_2 \sim \text{Unif}(0, 1)$ using a standard PRNG.
2. Let

$$\begin{aligned} Z_1 &= \sqrt{-2\log U_1}\cos(2\pi\, U_2) \\ Z_2 &= \sqrt{-2\log U_1}\sin(2\pi\, U_2) \end{aligned}$$

   Then Z_1 and Z_2 are distributed independent $N(0, 1)$.

What about multivariate Normal random variables with arbitrary covariance structure? This can be done by applying an affine transformation to independent Normals.

If we want to generate $X \sim \mathcal{N}(\mu, \Sigma)$, we can

1. Generate $Z \sim \mathcal{N}(0, I)$ where I is the identity matrix;
2. Let $\Sigma = LL'$ be the Cholesky decomposition of Σ.
3. Let $X = \mu + Lz$. Then $X \sim \mathcal{N}(\mu, \Sigma)$.

In reality, you will not need to apply *any* of the transformations described above because almost any worthwhile analytical software system will have these generators built in, if not carved in stone. However, once in a while, it's still nice to know how things work.

6.3 Rejection Sampling

What do we do if we want to generate samples of a random variable with density f and there isn't a built in function for doing this? If the random variable is of a reasonably low dimension (less than 10?), then *rejection sampling* is a plausible general approach.

The idea of rejection sampling is that although we cannot easily sample from f, there exists another density g, like a Normal distribution or perhaps a t-distribution, from which it is easy for us to sample (because there's a built in function or someone else wrote a nice function). Then we can sample from g directly and then "reject" the samples in a strategic way to make the resulting "non-rejected" samples look like they came from f. The density g will be referred to as the "candidate density" and f will be the "target density".

In order to use the rejections sampling algorithm, we must first ensure that the support of f is a subset of the support of g. If $\mathcal{X}_f$ is the support of f and $\mathcal{X}_g$ is the support of g, then we must have $\mathcal{X}_f \subset \mathcal{X}_g$. This makes sense: if there's a region of the support of f that g can never touch, then that area will never get sampled. In addition, we must assume that

$$c = \sup_{x \in \mathcal{X}_f} \frac{f(x)}{g(x)} < \infty$$

and that we can calculate c. The easiest way to satisfy this assumption is to make sure that g has heavier tails than f. We cannot have that g decreases at a faster rate than f in the tails or else rejection sampling will not work.

6.3.1 The Algorithm

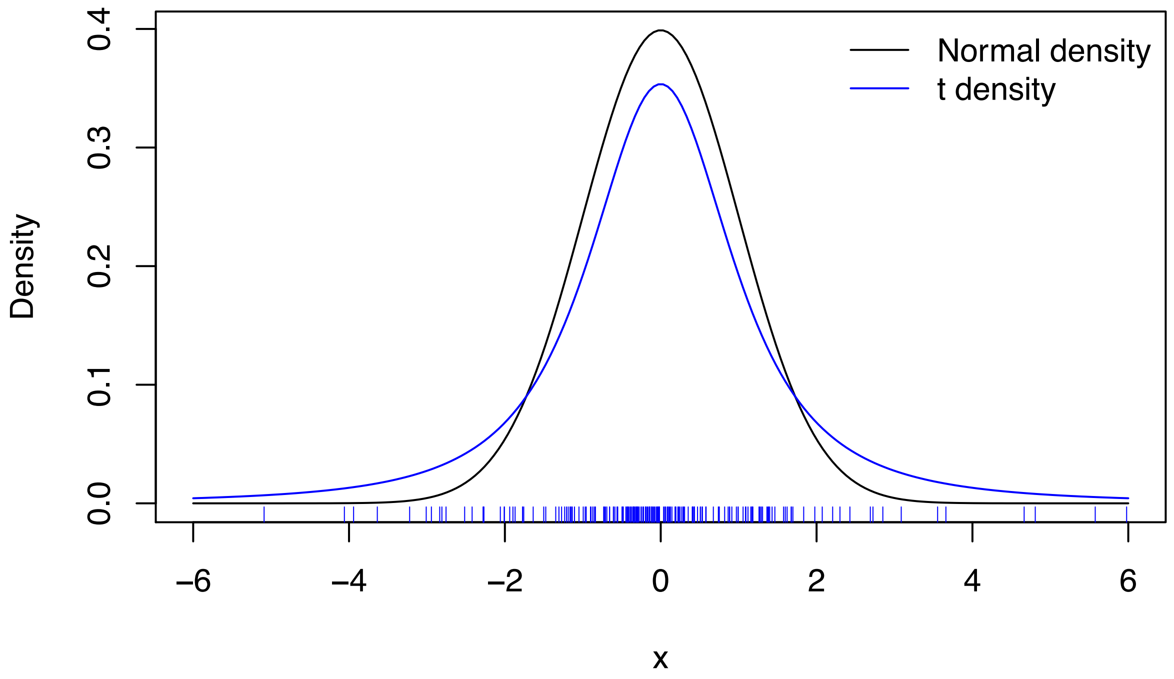
The rejection sampling algorithm for drawing a sample from the target density f is then

1. Simulate $U \sim \text{Unif}(0, 1)$.
2. Simulate a candidate $X \sim g$ from the candidate density
3. If
$$U \leq \frac{f(X)}{c\,g(X)}$$
then "accept" the candidate X. Otherwise, "reject" X and go back to the beginning.

The algorithm can be repeated until the desired number of samples from the target density f has been accepted.

As a simple example, suppose we wanted to generate samples from a $\mathcal{N}(0, 1)$ density. We could use the t_2 distribution as our candidate density as it has heavier tails than the Normal. Plotting those two densities, along with a sample from the t_2 density gives us the picture below.

```
set.seed(2017-12-4)
curve(dnorm(x), -6, 6, xlab = "x", ylab = "Density", n = 200)
curve(dt(x, 2), -6, 6, add = TRUE, col = 4, n = 200)
legend("topright", c("Normal density", "t density"),
       col = c(1, 4), bty = "n", lty = 1)
x <- rt(200, 2)
rug(x, col = 4)
```



Given what we know about the standard Normal density, most of the samples should be between −3 and +3, except perhaps in very large samples (this is a sample of size 200). From the picture, there are samples in the range of 4–6. In order to transform the t_2 samples into $\mathcal{N}(0, 1)$ samples, we will need to reject many of the samples out in the tail. On the other hand, there are two *few* samples in the range of $[-2, 2]$ and so we will have to disproportionaly accept samples in that range until it represents the proper $\mathcal{N}(0, 1)$ density.

Before we move on, it's worth noting that the rejection sampling method requires that we can *evaluate* the target density f. That is how we compute the rejection/acceptance ratio in Step 2. In most cases, this will not be a problem.

6.3.2 Properties of Rejection Sampling

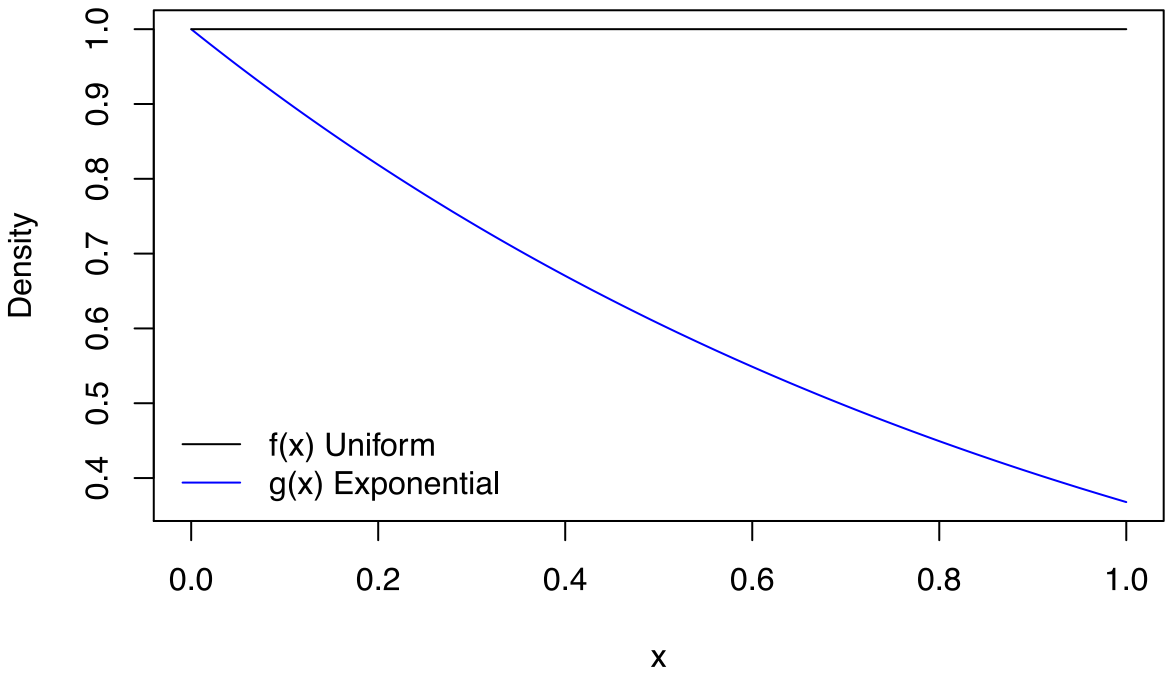
One property of the rejection sampling algorithm is that the number of draws we need to take from the candidate density g before we accept a candidate is a geometric random variable with success probability $1/c$. We can think of the decision to accept or reject a candidate as a sequence of iid coin flips that has a specific probability of coming up "heads" (i.e. being accepted). That probability is $1/c$ and we can calculate that as follows.

$$\begin{aligned}
\mathbb{P}(X \text{ accepted}) &= \mathbb{P}\left(U \leq \frac{f(X)}{c\,g(X)}\right) \\
&= \int \mathbb{P}\left(U \leq \frac{f(x)}{c\,g(x)}\,\middle|\, X = x\right) g(x)\,dx \\
&= \int \frac{f(x)}{c\,g(x)} g(x)\,dx \\
&= \frac{1}{c}
\end{aligned}$$

This property of rejection sampling has implications for how we choose the candidate density g. In theory, any density can be chosen as the candidate as long as its support includes the support of f. However, in practice we will want to choose g so that it matches f as closely as possible. As a rule of thumb, candidates g that match f closely will have smaller values of c and thus will accept candidates with higher probability. We want to avoid large values of c because large values of c lead to an algorithm that rejects a lot of candidates and has lower efficiency.

In the example above with the Normal distribution and the t_2 distribution, the ratio $f(x)/g(x)$ was maximized at $x = 1$ (or $x = -1$) and so the value of c for that setup was 1.257, which implies an acceptance probability of about 0.8. Suppose however, that we wanted to simulate from a Uniform$(0, 1)$ density and we used an Exponential(1) as our candidate density. The plot of the two densities looks as follows.

```r
curve(dexp(x, 1), 0, 1, col = 4, ylab = "Density")
segments(0, 1, 1, 1)
legend("bottomleft", c("f(x) Uniform", "g(x) Exponential"), lty = 1, col = c(1, 4), bty = "n")
```



Here, the ratio of $f(x)/g(x)$ is maximized at $x = 1$ and so the value of c is 2.718 which implies an acceptance probablity of about 0.37. While running the rejection sampling algorithm in this way to produce Uniform random variables will still work, it will be very inefficient.

We can now show that the distribution of the accepted values from the rejection sampling algorithm above follows the target density f. We can do this by calculating the *distribution function* of the accepted values and show that this is equal to $F(t) = \int_{-\infty}^{t} f(x)\,dx$.

$$\begin{aligned}
\mathbb{P}(X \leq t \mid X \text{ accepted}) &= \frac{\mathbb{P}(X \leq t, X \text{ accepted})}{\mathbb{P}(X \text{ accepted})} \\
&= \frac{\mathbb{P}(X \leq t, X \text{ accepted})}{1/c} \\
&= c\,\mathbb{E}_g\mathbb{E}\left[\mathbf{1}\{x \leq t\}\mathbf{1}\left\{U \leq \frac{f(x)}{c\,g(x)}\right\}\middle| X = x\right] \\
&= c\,\mathbb{E}_g\left[\mathbf{1}\{X \leq t\}\mathbb{E}\left[\mathbf{1}\left\{U \leq \frac{f(x)}{c\,g(x)}\right\}\middle| X = x\right]\right] \\
&= c\,\mathbb{E}_g\left[\mathbf{1}\{X \leq t\}\frac{f(X)}{c\,g(X)}\right] \\
&= \int_{-\infty}^{\infty} \mathbf{1}\{x \leq t\}\frac{f(x)}{g(x)}g(x)\,dx \\
&= \int_{-\infty}^{t} f(x)\,dx \\
&= F(t)
\end{aligned}$$

This shows that the distribution function of the candidate values, given that they are accepted, is equal to the distribution function corresponding to the target density.

A few further notes:

1. We only need to know f and g up to a constant of proportionality. In many applications we will not know the normalizing constant for these densities, but we do not need them. That is, if $f(x) = k_1 f^{\star}(x)$

and $g(x) = k_2 g^\star(x)$, we can proceed with the algorithm using $f^\star$ and $g^\star$ even if we do not know the values of k_1 and k_2.

2. Any number $c' \geq c$ will work in the rejection sampling algorithm, but the algorithm will be less efficient.
3. Throughout the algorithm, operations can (and should!) be done on a log scale.
4. The higher the dimension of f and g, the less efficient the rejection sampling algorithm will be.
5. Whether $c = \infty$ or not depends on the tail behavior of the the densities f and g. If $g(x) \downarrow 0$ faster than $f(x) \downarrow 0$ as $x \to \infty$, then $f(x)/g(x) \uparrow \infty$.

6.3.3 Empirical Supremum Rejection Sampling

What if we cannot calculate $c = \sup_{x \in \mathcal{X}_f} \frac{f(x)}{g(x)}$ or are simply too lazy to do so? Fear not, because it turns out we almost never have to do so. A slight modification of the standard rejection sampling algorithm will allow us to *estimate* c while also sampling from the target density f. The tradeoff (there is always a tradeoff!) is that we must make a more stringent assumption about c, mainly that it is achievable. That is, there exists some value $x_c \in \mathcal{X}_f$ such that $\frac{f(x_c)}{g(x_c)}$ is *equal* to $\sup_{x \in \mathcal{X}_f} \frac{f(x)}{g(x)}$.

The modified algorithm is the *empirical supremum rejection sampling* algorithm of Caffo, Booth, and Davison. The algorithm goes as follows. First we must choose some starting value of c, call it $\hat{c}$, such that $\hat{c} > 1$. Then,

1. Draw $U \sim \text{Unif}(0, 1)$.
2. Draw $X \sim g$, the candidate density.
3. Accept X if $U \leq \frac{f(X)}{\hat{c}\, g(X)}$, otherwise reject X.
4. Let $\hat{c}^\star = \max\left\{\hat{c}, \frac{f(X)}{g(X)}\right\}$.
5. Update $\hat{c} = \hat{c}^\star$.
6. Goto Step 1.

From the algorithm we can see that at each iteration, we get more information about the ratio $f(X)/g(X)$ and can update our estimate of c accordingly.

One way to think of this algorithm is to conceptualize a separate sequence $\tilde{Y}_i$, which is 0 or 1 depending on whether X_i should be rejected (0) or accepted (1). This sequence $\tilde{Y}_i$ is the accept/reject determination sequence. Under the standard rejection sampling algorithm, the sequence $\tilde{Y}_i$ is generated using the true value of c. Under the emprical supremum rejection sampling (ESUP) scheme, we generate a slightly different sequence Y_i using our continuously updated value of $\hat{c}$.

If we drew values $X_1, X_2, X_3, X_4, X_5, X_6, \ldots$ from the candidate density g, then we could visualize the acceptance/rejection process as it might occur using the true value of c and our estimate $\hat{c}$.

Following the diagram above, we can see that using the estimate $\hat{c}$, there are two instances where we accept a value when we should have rejected it (X_1 and X_4). In every other instance in the sequence, the value of Y_i was equal to $\tilde{Y}_i$. The theory behind the ESUP algorithm is that eventually, the sequence Y_i becomes identical to the sequence $\tilde{Y}_i$ and therefore we will accept/reject candidates in the same manner as we would have if we had used the true c.

If f and g are discrete distributions, then the proof of the ESUP algorithm is fairly straightforward. Specifically, Caffo, Booth, and Davison showed that $\mathbb{P}(Y_i \neq \tilde{Y}_i \text{ infinitely often}) = 0$. Recall that by assumption, there exists some $x_c \in \mathcal{X}_f$ such that $c = \frac{f(x_c)}{g(x_c)}$. Therefore, as we independently sample candidates from g, at *some point*, we will sample the value x_c, in which case we will achieve the value c. Once that happens, we are then using the standard rejection sampling algorithm and our estimate $\hat{c}$ never changes.

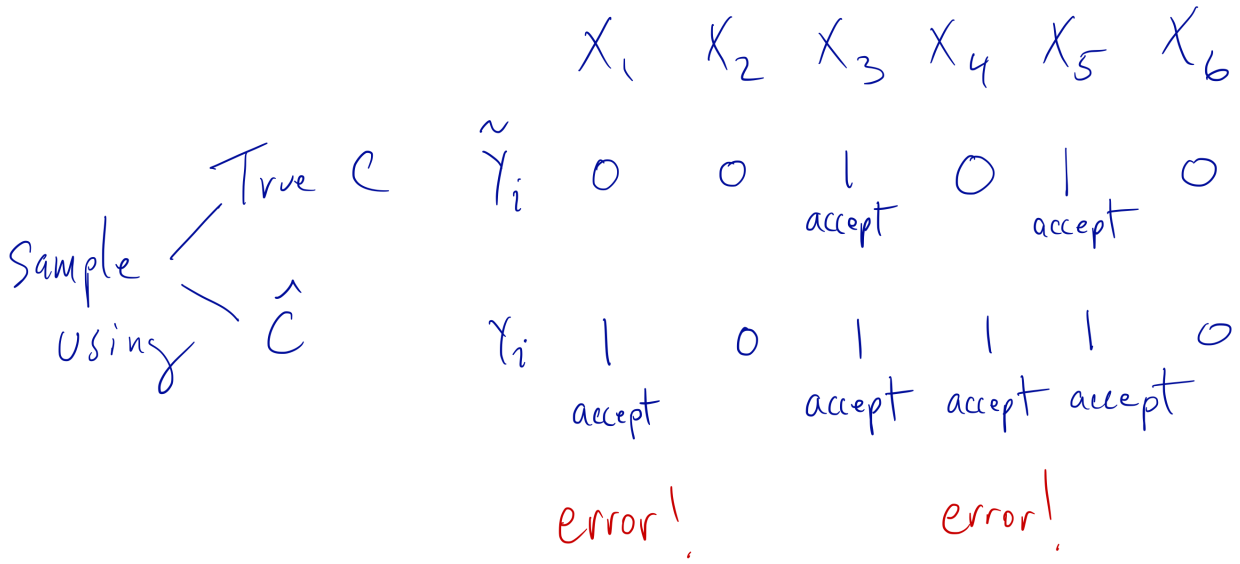


Figure 8: Empirical supremum rejection sampling scheme.

Let $\gamma = \min_i\{x_i = x_c\}$, where $x_i \sim g$. So γ is the first time that we see the value x_c as we are sampling candidates x_i from g. The probability that we sample x_c is $g(x_c)$ (recall that g is assumed to be discrete here) and so γ has a Geometric distribution with success probability $g(x_c)$. Once we observe x_c, the ESUP algorithm and the standard rejection sampling algorithms converge and are identical.

From here, we can use the coupling inequality, which tells us that

$$\mathbb{P}(Y_i \neq \tilde{Y}_i) \leq \mathbb{P}(\gamma \geq i).$$

Given that $\gamma \sim \text{Geometric}(g(x_c))$, we know that

$$\mathbb{P}(\gamma \geq i) = (1 - g(x_c))^{i-1}.$$

This then implies that

$$\sum_{i=1}^{\infty} \mathbb{P}(Y_i \neq \tilde{Y}_i) < \infty$$

which, by the Borel-Cantelli lemma, implies that $\mathbb{P}(Y_i \neq \tilde{Y}_i \text{ infinitely often}) = 0$. Therefore, eventually the sequences Y_i and $\tilde{Y}_i$ must converge and at that point the ESUP algorithm will be identical to the rejection sampling algorithm.

In practice, we will know know exactly when the ESUP algorithm has converged to the standard rejection sampling algorithm. However, Caffo, and Davison report that the convergence is generally fast. Therefore, a reasonable approach might be to discard the first several accepted values (e.g. a "burn in") and then use the remaining values.

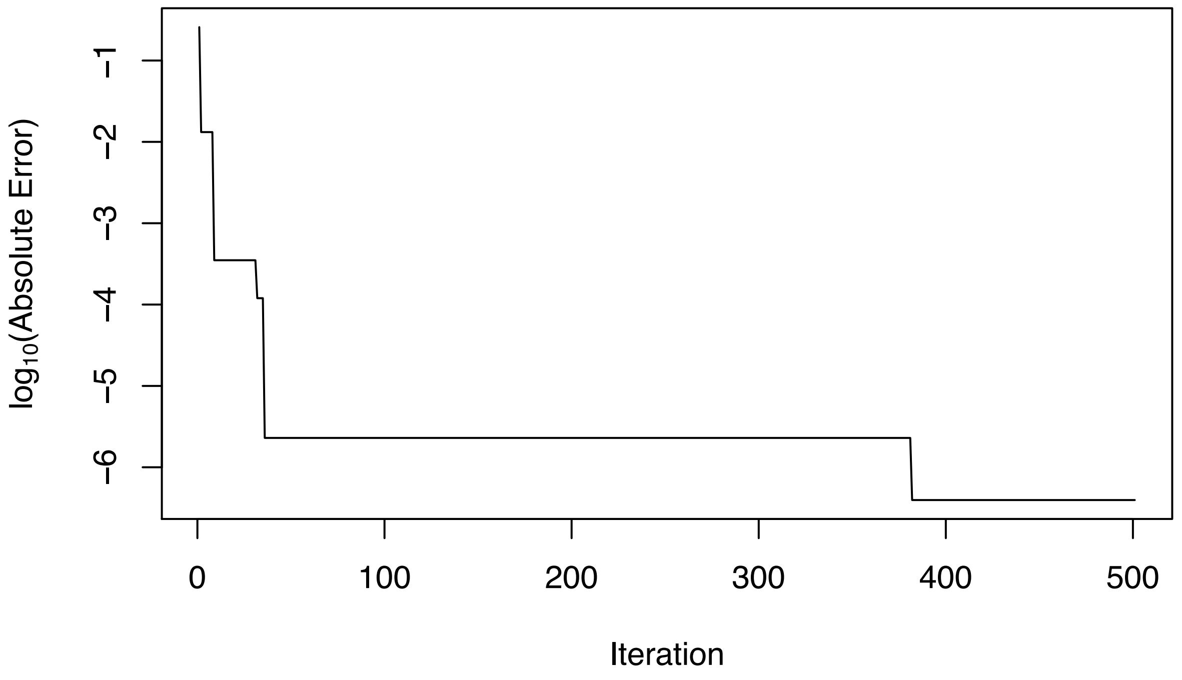
We can see how quickly ESUP converges in a simple example where the target density is the standard Normal and the candidate density is the t_2 distribution. Here we simulate 1,000 draws and start with a value $\hat{c} = 1.0001$. Note that in the code below, all of the computations are done on the log scale for the sake of numerical stability.

```r
set.seed(2017-12-04)
N <- 500
y_tilde <- numeric(N)  ## Binary accept/reject for "true" algorithm
y <- numeric(N)        ## Binary accept/reject for ESUP
```

```
log_c_true <- dnorm(1, log = TRUE) - dt(1, 2, log = TRUE)
log_chat <- numeric(N + 1)
log_chat[1] <- log(1.0001)  ## Starting c value
for(i in seq_len(N)) {
        u <- runif(1)
        x <- rt(1, 2)
        r_true <- dnorm(x, log = TRUE) - dt(x, 2, log = TRUE) - log_c_true
        rhat <- dnorm(x, log = TRUE) - dt(x, 2, log = TRUE) - log_chat[i]
        y_tilde[i] <- log(u) <= r_true
        y[i] <- log(u) <= rhat
        log_chat[i+1] <- max(log_chat[i],
                             dnorm(x, log = TRUE) - dt(x, 2, log = TRUE))
}
```

Now we can plot $\log_{10}(|\hat{c} - c|)$ for each iteration to see how the magnitude of the error changes with each iteration.

```
c_true <- exp(log_c_true)
chat <- exp(log_chat)
plot(log10(abs(chat - c_true)), type = "l",
     xlab = "Iteration", ylab = expression(paste(log[10], "(Absolute Error)")))
```



We can see that by iteration 40 or so, $\hat{c}$ and c differ only in the 5th decimal place and beyond. By the 380th iteration, they differ only beyond the 6th decimal place.

6.4 Importance Sampling

With rejection sampling, we ultimately obtain a sample from the target density f. With that sample, we can create any number of summaries, statistics, or visualizations. However, what if we are interested in the more narrow problem of computing a mean, such as $\mathbb{E}_f[h(X)]$ for some function $h : \mathbb{R}^k \to \mathbb{R}$? Clearly, this is a problem that can be solved with rejection sampling: First obtain a sample $x_1, \dots, x_n \sim f$ and then compute

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^n h(x_i).$$

with the obtained sample. As $n \to \infty$ we know by the Law of Large Numbers that $\hat{\mu}_n \to \mathbb{E}_f[h(X)]$. Further, the Central Limit Theorem gives us $\sqrt{n}(\hat{\mu}_n - \mathbb{E}_f[h(X)]) \longrightarrow \mathcal{N}(0, \sigma^2)$. So far so good.

However, with rejection sampling, in order to obtain a sample of size n, we must generate, on average, $c \times n$ candidates from g, the candidate density, and then reject about $(c-1) \times n$ of them. If $c \approx 1$ then this will not be too inefficient. But in general, if c is much larger than 1 then we will be generating a lot of candidates from g and ultimately throwing most of them away.

It's worth noting that in most cases, the candidates generated from g fall within the domain of f, so that they are in fact values that could plausibly come from f. They are simply over- or under-represented in the frequency with which they appear. For example, if g has heavier tails than f, then there will be too many extreme values generated from g. Rejection sampling simply thins out those extreme values to obtain the right proportion. But what if we could take those rejected values and, instead of discarding them, simply downweight or upweight them in a specific way?

Note that we can rewrite the target estimation as follows,

$$\mathbb{E}_f[h(X)] = \mathbb{E}_g\left[\frac{f(X)}{g(X)}h(X)\right].$$

Hence, if $x_1, \dots, x_n \sim g$, drawn from the candidate density, we can say

$$\tilde{\mu}_n = \frac{1}{n}\sum_{i=1}^n \frac{f(x_i)}{g(x_i)}h(x_i) = \frac{1}{n}\sum_{i=1}^n w_i h(x_i) \approx \mathbb{E}_f[h(X)]$$

In the equation above, the values $w_i = f(x_i)/g(x_i)$ are referred to as the *importance weights* because they take each of the candidates x_i generated from g and reweight them when taking the average. Note that if $f = g$, so that we are simply sampling from the target density, then this estimator is just the sample mean of the $h(x_i)$s. The estimator $\tilde{\mu}_n$ is known as the *importance sampling* estimator.

When comparing rejection sampling with importance sampling, we can see that

- Rejection sampling samples directly from f and then uses the samples to compute a simple mean
- Importance sampling samples from g and then reweights those samples by $f(x)/g(x)$

For estimating expectations, one might reasonably believe that the importance sampling approach is more efficient than the rejection sampling approach because it does not discard any data.

In fact, we can see this by writing the rejection sampling estimator of the expectation in a different way. Let $c = \sup_{x \in \mathcal{X}_f} f(x)/g(x)$. Given a sample $x_1, \dots, x_n \sim g$ and $u_1, \dots, u_n \sim \text{Unif}(0, 1)$, then

$$\hat{\mu}_n = \frac{\sum_i \mathbf{1}\left\{u_i \le \frac{f(x_i)}{c\,g(x_i)}\right\} h(x_i)}{\sum_i \mathbf{1}\left\{u_i \le \frac{f(x_i)}{c\,g(x_i)}\right\}}$$

What importance sampling does, effectively, is replace the indicator functions in the above expression with their expectation. So instead of having a hard threshold, where observation x_i is either included (accepted) or not (rejected), importance sampling smooths out the acceptance/rejection process so that every observation plays some role.

If we take the expectation of the indicator functions above, we get (note that the cs cancel)

$$\tilde{\mu}_n = \frac{\sum_i \frac{f(x_i)}{g(x_i)} h(x_i)}{\sum_i \frac{f(x_i)}{g(x_i)}} = \frac{\frac{1}{n}\sum_i \frac{f(x_i)}{g(x_i)} h(x_i)}{\frac{1}{n}\sum_i \frac{f(x_i)}{g(x_i)}}$$

which is roughly equivalent to the importance sampling estimate if we take into account that

$$\frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{g(x_i)} \approx 1$$

because

$$\mathbb{E}_g\left[\frac{f(X)}{g(X)}\right] = \int \frac{f(x)}{g(x)} g(x)\,dx = 1$$

The point of all this is to show that the importance sampling estimator of the mean can be seen as a "smoothed out" version of the rejection sampling estimator. The advantage of the importance sampling estimator is that it does not discard any data and thus is more efficient.

Note that we do not need to know the normalizing constants for the target density or the candidate density. If $f^\star$ and $g^\star$ are the unnormalized target and candidate densities, respectively, then we can use the modified importance sampling estimator,

$$\mu_n^\star = \frac{\sum_i \frac{f^\star(x_i)}{g^\star(x_i)} h(x_i)}{\sum_i \frac{f^\star(x_i)}{g^\star(x_i)}}.$$

We can then use Slutsky's Theorem to say that $\mu_n^\star \to \mathbb{E}_f[h(X)]$.

6.4.1 Example: Bayesian Sensitivity Analysis

An interesting application of importance sampling is the examination of the sensitivity of posterior inferences with respect to prior specification. Suppose we observe data y with density $f(y \mid \theta)$ and we specify a prior for θ as $\pi(\theta \mid \psi_0)$, where ψ_0 is a hyperparameter. The posterior for θ is thus

$$p(\theta \mid y, \psi_0) \propto f(y \mid \theta)\pi(\theta \mid \psi_0)$$

and we would like to compute the posterior mean of θ. If we can draw $\theta_1, \dots, \theta_n$, a sample of size n from $p(\theta \mid y, \psi_0)$, then we can estimate the posterior mean with $\frac{1}{n}\sum_i \theta_i$. However, this posterior mean is estimated using a specific hyperparameter ψ_0. What if we would like to see what the posterior mean would be for a different value of ψ? Do we need to draw a new sample of size n? Thankfully, the answer is no. We can simply take our existing sample $\theta_1, \dots, \theta_n$ and reweight it to get our new posterior mean under a different value of ψ.

Given a sample $\theta_1, \dots, \theta_n$ drawn from $p(\theta \mid y, \psi_0)$, we would like to know $\mathbb{E}[\theta \mid y, \psi]$ for some $\psi \neq \psi_0$. The idea is to treat our original $p(\theta \mid y, \psi_0)$ as a "candidate density" from which we have already drawn a large sample $\theta_1, \dots, \theta_n$. Then we want know the posterior mean of θ under a "target density" $p(\theta \mid y, \psi)$. We can then write our importance sampling estimator as

$$\begin{aligned}
\frac{\sum_i \theta_i \frac{p(\theta_i|y,\psi)}{p(\theta_i|y,\psi_0)}}{\sum_i \frac{p(\theta_i|y,\psi)}{p(\theta_i|y,\psi_0)}} &= \frac{\sum_i \theta_i \frac{f(y|\theta_i)\pi(\theta_i|\psi)}{f(y|\theta_i)\pi(\theta_i|\psi_0)}}{\sum_i \frac{f(y|\theta_i)\pi(\theta_i|\psi)}{f(y|\theta_i)\pi(\theta_i|\psi_0)}} \\
&= \frac{\sum_i \theta_i \frac{\pi(\theta_i|\psi)}{\pi(\theta_i|\psi_0)}}{\sum_i \frac{\pi(\theta_i|\psi)}{\pi(\theta_i|\psi_0)}} \\
&\approx \mathbb{E}[\theta \mid y, \psi]
\end{aligned}$$

In this case, the importance sampling weights are simply the ratio of the prior under ψ to the prior under ψ_0.

6.4.2 Properties of the Importance Sampling Estimator

So far we've talked about how to estimate an expectation with respect to an arbitrary target density f using importance sampling. However, we haven't discussed yet what is the variance of that estimator. An analysis of the variance of the importance sampling estimator is assisted by the Delta method and by viewing the importance sampling estimator as a ratio estimator.

Recall that the Delta method states that if Y_n is a k-dimensional random variable with mean μ, $g : \mathbb{R}^k \to \mathbb{R}$ and is differentiable, and further we have

$$\sqrt{n}(Y_n - \mu) \xrightarrow{D} \mathcal{N}(0, \Sigma)$$

as $n \to \infty$, then

$$\sqrt{n}(g(Y_n) - g(\mu)) \xrightarrow{D} \mathcal{N}(0, g'(\mu)'\Sigma g'(\mu))$$

as $n \to \infty$.

For the importance sampling estimator, we have f is the target density, g is the candidate density, and $x_1, \dots, x_n$ are samples from g. The estimator of $\mathbb{E}_f[h(X)]$ is written as

$$\frac{\frac{1}{n}\sum_i h(x_i)w(x_i)}{\frac{1}{n}\sum_i w(x_i)}$$

where

$$w(x_i) = \frac{f(x_i)}{g(x_i)}$$

are the importance sampling weights.

If we let $g((a,b)) = a/b$, then $g'((a,b)) = (1/b, -a/b^2)$. If we define the vector $Y_n = \left(\frac{1}{n}\sum h(x_i)w_i,\ \frac{1}{n}\sum w_i\right)$ then the importance sampling estimator is simply $g(Y_n)$. Furthremore, we have

$$\mathbb{E}_g[Y_n] = \mathbb{E}_g\left[\left(\frac{1}{n}\sum h(x_i)w(x_i),\ \frac{1}{n}\sum w(x_i)\right)\right] = (\mathbb{E}_f[h(X)], 1) = \mu$$

and

$$\Sigma = n\,\mathrm{Var}(Y_n) = \begin{pmatrix} \mathrm{Var}(h(X)w(X)) & \mathrm{Cov}(h(X)w(X), w(X)) \\ \mathrm{Cov}(h(X)w(X), w(X)) & \mathrm{Var}(w(X)) \end{pmatrix}$$

Note that the above quantity can be estimated consistently using the sample versions of each quantity in the matrix.

Therefore, the variance of the importance sampling estimator of $\mathbb{E}_f[h(X)]$ is $g'(Y_n)'\Sigma g'(Y_n)$ which we can expand to

$$n\left(\frac{\sum h(x_i)w(x_i)}{\sum w(x_i)}\right)^2 \left(\frac{\sum h(x_i)^2 w(x_i)^2}{\left(\sum h(x_i)w(x_i)\right)^2} - 2\frac{\sum h(x_i)w(x_i)^2}{\left(\sum h(x_i)w(x_i)\right)\left(\sum w(x_i)\right)} + \frac{\sum w(x_i)^2}{\left(\sum w(x_i)\right)^2}\right)$$

Given this, for the importance sampling estimator, we need the following to be true,

$$\mathbb{E}_g\left[h(X)^2 w(X)^2\right] = \mathbb{E}_g\left[h(X)\frac{f(X)}{g(X)}\right] < \infty,$$

$$\mathbb{E}_g[w(X)^2] = \mathbb{E}_g\left[\left(\frac{f(X)}{g(X)}\right)^2\right] < \infty,$$

and

$$\mathbb{E}_g\left[h(X)w(X)^2\right] = \mathbb{E}_g\left[h(X)\left(\frac{f(X)}{g(X)}\right)^2\right] < \infty.$$

All of the above conditions are true if the conditions for rejection sampling are satisfied, that is, if $\sup_{x\in\mathcal{X}_f} \frac{f(x)}{g(x)} < \infty$.

7 Markov Chain Monte Carlo

The phrase "Markov chain Monte Carlo" encompasses a broad array of techniques that have in common a few key ideas. The setup for all the techniques that we will discuss in this book is as follows:

1. We want to sample from a some complicated density or probability mass function π. Often, this density is the result of a Bayesian computation so it can be interpreted as a posterior density. The presumption here is that we can *evaluate* π but we cannot *sample* from it.
2. We know that certain stochastic processes called *Markov chains* will converge to a stationary distribution (if it exists and if specific conditions are satisfied). Simulating from such a Markov chain for a long enough time will eventually give us a sample from the chain's stationary distribution.
3. Given the functional form of the density π, we want to construct a Markov chain that has π as its stationary distribution.
4. We want to sample values from the Markov chain such that the sequence of values $\{x_n\}$ generated by the chain converges in distribution to the density π.

In order for all these ideas to make sense, we need to first go through some background on Markov chains. The rest of this chapter will be spent defining all these terms, the conditions under which they make sense, and giving examples of how they can be implemented in practice.

7.1 Background

There are many introductions and overviews of Markov chain Monte Carlo out there and a quick web search will reveal them. A decent introductory book is Markov Chain Monte Carlo in Practice by Gilks, Richardson, and Spiegelhalter, but there are many others. In this section we will give just the briefest of overviews to get things started.

A Markov chain is a stochastic process that evolves over time by transitioning into different states. The sequence of states is denoted by the collection $\{X_i\}$ and the transition between states is random, following the rule

$$\mathbb{P}(X_t \mid X_{t-1}, X_{t-2}, \dots, X_0) = \mathbb{P}(X_t \mid X_{t-1})$$

(The above notation is for discrete distributions; we will get to the continuous case later.)

This relationship means that the probability distribution of the process at time t, given all of the previous values of the chain, is equal to the probability distribution given just the previous value (this is also known as the Markov property). So in determining the sequence of values that the chain takes, we can determine the distribution of our next value given just our current value.

The collection of states that a Markov chain can visit is called the *state space* and the quantity that governs the probability that the chain moves from one state to another state is the *transition kernel* or *transition matrix*.

7.1.1 A Simple Example

A classic example of a Markov chain has a state space and transition probabilities that can be drawn as follows.

Here, we have three states in the state space and the arrows indicate to which state you can travel given your current state. The fractions next to each arrow tell you what is the transition probabilities of going to a given state. Note that in this Markov chain, you can only ever travel to two possible states no matter what your current state happens to be.

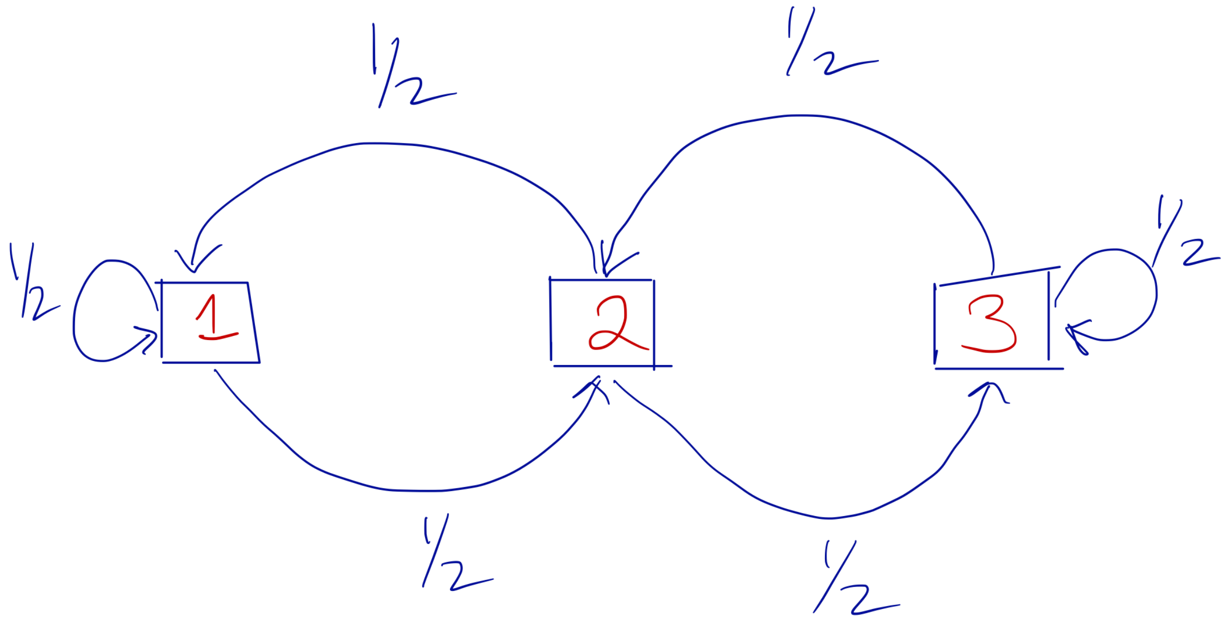


Figure 9: Simple Markov chain

We can write the transition probabilities in a matrix, a.k.a. the transition matrix, as

$$P = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1/2 & 1/2 \end{bmatrix}$$

With the matrix notation, the interpretation of the entries is that if we are currently on interation n of the chain, then

$$\mathbb{P}(X_{n+1} = j \mid X_n = i) = P_{ij}$$

From the matrix, we can see that the probability of going from state 1 to state 3 is 0, because the $(1, 3)$ entry of the matrix is 0.

Suppose we start this Markov chain in state 3 with probability 1, so the initial probability distribution over the three states is $\pi_0 = (0, 0, 1)$. What is the probability distribution of the states after one iteration? The way the transition matrix works, we can write

$$\pi_1 = \pi_0 P = (0, 1/2, 1/2)$$

A quick check of the diagram above confirms that if we start in state 3, then after one iteration we can only be in either state 3 or in state 2. So state 1 gets 0 probability.

What is the probability distribution over the states after n iterations? We can continue the process above to get

$$\pi_n = \pi_0 \overbrace{PPPP \cdots P}^{n \text{ times}} = P^{(n)}$$

For example, after five iterations, starting in state 3, we would have $\pi_5 = (0.3125, 0.34375, 0.34375)$.

7.1.2 Basic Limit Theorem

For a Markov chain with a discrete state space and transition matrix P, let $\pi_\star$ be such that $\pi_\star P = \pi_\star$. Then $\pi_\star$ is a *stationary distribution* of the Markov chain and the chain is said to be stationary if it reaches this distribution.

The basic limit theorem for Markov chains says that, under a specific set of assumptions that we will detail below, we have

$$\|\pi_\star - \pi_n\| \longrightarrow 0$$

as $n \to \infty$, where $\|\cdot\|$ is the total variation distance between the two densities. Therefore, no matter where we start the Markov chain (π_0), π_n will eventually approach the stationary distribution. Another way to think of this is that

$$\lim_{n\to\infty} \pi_n(i) = \pi_\star(i)$$

for all states i in the state space.

There are three assumptions that we must make in order for the basic limit theorem to hold.

1. The stationary distribution $\pi_\star$ exists.
2. The chain is **irreducible**. The basic idea with irreducibility is that there should be a path between any two states and that path should altogether have probability > 0. This does not mean that you can transition to any state from any other state *in one step*. Rather, there is a sequence of steps that can be taken with positive probability that would connect any two states. Mathematically, we can write this as there exists some n such that $\mathbb{P}(X_n = j \mid X_0 = i)$ for all i and j in the state space.
3. The chain is **aperiodic**. A chain is aperiodic if it does not make deterministic visits to a subset of the state space. For example, consider a Markov chain on the space of integers that starts at $X_0 = 0$, and has a transition rule where

   $$X_{n+1} = X_n + \varepsilon_n$$

   and

   $$\varepsilon_n = \begin{cases} 1 & \text{w/prob } \frac{1}{2} \\ -1 & \text{w/prob } \frac{1}{2} \end{cases}$$

   This chain clearly visits the odd numbers when n is odd and the even numbers when n is even. Furthermore, if we are at any state i, we cannot revisit state i except in a number of steps that is a multiple of 2. This chain therefore has a period of 2. If a chain has a period of 1 it is aperiodic (otherwise it is periodic).

7.1.3 Time Reversibility

A Markov chain is *time reversible* if

$$(X_0, X_1, \dots, X_n) \overset{D}{=} (X_n, X_{n-1}, \dots, X_0).$$

The sequence of states moving in the "forward" direction (with respect to time) is equal in distribution to the sequence of states moving in the "backward" direction. Further, the definition above implies that $(X_0, X_1) \overset{D}{=} (X_1, X_0)$, which further implies that $X_0 \overset{D}{=} X_1$. Because X_1 is equal in distribution to X_0, this implies that $\pi_1 = \pi_0$. However, because $\pi_1 = \pi_0 P$, where P is the transition matrix, this means that π_0 is the stationary distribution, which we will now refer to as π. Therefore, a time reversible Markov chain is stationary.

In addition to stationarity, the time reversibility property tells us that for all i, j, the following are all equivalent:

$$\begin{aligned}
(X_0, X_1) &\overset{D}{=} (X_1, X_0) \\
\mathbb{P}(X_0 = i, X_1 = j) &= \mathbb{P}(X_1 = i, X_0 = j) \\
\mathbb{P}(X_0 = i)\mathbb{P}(X_1 = j \mid X_0 = i) &= \mathbb{P}(X_0 = j)\mathbb{P}(X_1 = i \mid X_0 = j)
\end{aligned}$$

The last line can also be written as

$$\pi(i)P(i,j) = \pi(j)P(j,i)$$

which are called the *local balance equations*. A key property that we will exploit later is that if the local balance equations hold for a transition matrix P and distribution π, then π is the stationary distribution of a chain governed by the transition matrix P.

Why is all this important? Time reversibility is relevant to analyses making use of Markov chain Monte Carlo because it allows us to find a way to construct a proper Markov chain from which to simulate. In most MCMC applications, we have no trouble identifying the stationary distribution. We already know the stationary distribution because it is usually a posterior density resulting from a complex Bayesian calculation. Our problem is that we cannot simulate from the distribution. So the problem is constructing a Markov chain that leads to a given stationary distribution.

Time reversibility gives us a way to construct a Markov chain that converges to a given stationary distribution. As long as we can show that a Markov chain with a given transition kernel/matrix satisfies the local balance equations with respect to the stationary distribution, we can know that the chain will converge to the stationary distribution.

7.1.4 Summary

To summarize what we have learned in this section, we have that

1. We want to sample from a complicated density π.
2. We know that aperiodic and irreducible Markov chains with a stationary distribution π will eventually converge to that stationary distribution.
3. We know that if a Markov chain with transition matrix P is time reversibile with respect to π then π must be the stationary distribution of the Markov chain.
4. Given a chain governed by transition matrix P, we can simulate it for a long time and eventually we will be simulating from π.

That is the basic gist of MCMC techniques and I have given the briefest of introductions here. That said, in the next section we will discuss how to construct a proper Markov chain for simulating from the stationary distribution.

7.2 Metropolis-Hastings

Let $q(Y \mid X)$ be a transition density for p-dimensional X and Y from which we can easily simulate and let $\pi(X)$ be our target density (i.e. the stationary distribution that our Markov chain will eventually converge to). The Metropolis-Hastings procedure is an iterative algorithm where at each stage, there are three steps. Suppose we are currently in the state x and we want to know how to move to the next state in the state space.

1. Simulate a *candidate* value $y \sim q(Y \mid x)$. Note that the candidate value depends on our current state x.
2. Let
$$\alpha(y \mid x) = \min\left\{\frac{\pi(y)q(x \mid y)}{\pi(x)q(y \mid x)}, 1\right\}$$
$\alpha(y \mid x)$ is referred to as the *acceptance ratio.*
3. Simulate $u \sim \text{Unif}(0,1)$. If $u \leq \alpha(y \mid x)$, then the next state is equal to y. Otherwise, the next state is still x (we stay in the same place).

This three step process represents the transition kernel for our Markov chain from which we are simulating. Recall that the hope is that our Markov chain will, after many simulations, converge to the stationary distribution. Eventually, we can be reasonably sure that the samples that we draw from this process are draws from the stationary distribution, i.e. $\pi(X)$.

Why does this work? Recall that we need to have the Markov chain generated by this transition kernel be time reversible. If $K(y \mid x)$ is the transition kernel embodied by the three steps above, then we need to show that

$$\pi(y)K(x \mid y) = \pi(x)K(y \mid x).$$

We can write the transition kernel $K(y \mid x)$ as

$$K(y \mid x) = \alpha(y \mid x)q(y \mid x) + \mathbf{1}\{y = x\}\left[1 - \int \alpha(s \mid x)q(s \mid x)\,ds\right]$$

The function $K(y \mid x)$ can be decomposed into two parts and we can treat each separately. First we can show that

$$\begin{aligned}
K(y \mid x)\pi(x) &= K(x \mid y)\pi(y)\\
\alpha(y \mid x)q(y \mid x)\pi(x) &= \alpha(x \mid y)q(x \mid y)\pi(y)\\
\min\left\{\frac{\pi(y)}{\pi(x)}\frac{q(x \mid y)}{q(y \mid x)}, 1\right\} q(y \mid x)\pi(x) &= \min\left\{\frac{\pi(x)}{\pi(y)}\frac{q(y \mid x)}{q(x \mid y)}, 1\right\} q(x \mid y)\pi(y)\\
\min(\pi(y)q(x \mid y),\, q(y \mid x)\pi(x)) &= \min(\pi(x)q(y \mid x),\, q(x \mid y)\pi(y))
\end{aligned}$$

The last line is trivially true because on both sides of the equation we are taking the minimum of the same quantities.

For the second part, let $r(x) = \left[1 - \int \alpha(s \mid x)q(s \mid x)\,ds\right]$. Then we need to show that

$$\mathbf{1}\{y = x\}\, r(x)\pi(x) = \mathbf{1}\{x = y\}\, r(y)\pi(y)$$

over the set where $y = x$. But this is trivially true because if $y = x$ then every quantity in the above equation is the same regardless of an x or a y appears in it.

7.2.1 Random Walk Metropolis-Hastings

Let $q(y \mid x)$ be defined as

$$y = x + \varepsilon$$

where $\varepsilon \sim g$ and g is a probability density symmetric about 0. Given this definition, we have

$$q(y \mid x) = g(\varepsilon)$$

and

$$q(x \mid y) = g(-\varepsilon) = g(\varepsilon)$$

Because $q(y \mid x)$ is symmetric in x and y, the Metropolis-Hastings acceptance ratio $\alpha(y \mid x)$ simplifies to

$$\begin{aligned}\alpha(y \mid x) &= \min\left\{\frac{\pi(y)q(x \mid y)}{\pi(x)q(y \mid x)}, 1\right\}\\ &= \min\left\{\frac{\pi(y)}{\pi(x)}, 1\right\}\end{aligned}$$

Given our current state x, the random walk Metropolis-Hastings algorithm proceeds as follows:

1. Simulate $\varepsilon \sim g$ and let $y = x + \varepsilon$.
2. Compute $\alpha(y \mid x) = \min\left\{\frac{\pi(y)}{\pi(x)}, 1\right\}$
3. Simulate $u \sim \text{Unif}(0, 1)$. If $u \leq \alpha(y \mid x)$ then accept y as the next state, otherwise stay at x.

It should be noted that this form of the Metropolis-Hastings algorithm was the original form of the *Metropolis algorithm*.

7.2.1.1 Example: Sampling Normal Variates

As a simple example, we can show how random walk Metropolis-Hastings can be used to sample from a standard Normal distribution. Let g be a uniform distribution over the interval $(-\delta, \delta)$, where δ is small and > 0 (its exact value doesn't matter). Then we can do

1. Simulate $\varepsilon \sim \text{Unif}(-\delta, \delta)$ and let $y = x + \varepsilon$.
2. Compute $\alpha(y \mid x) = \min\left\{\frac{\varphi(y)}{\varphi(x)}, 1\right\}$ where φ is the standard Normal density.
3. Simulate $u \sim \text{Unif}(0, 1)$. If $u \leq \alpha(y \mid x)$ then accept y as the next state, otherwise stay at x.

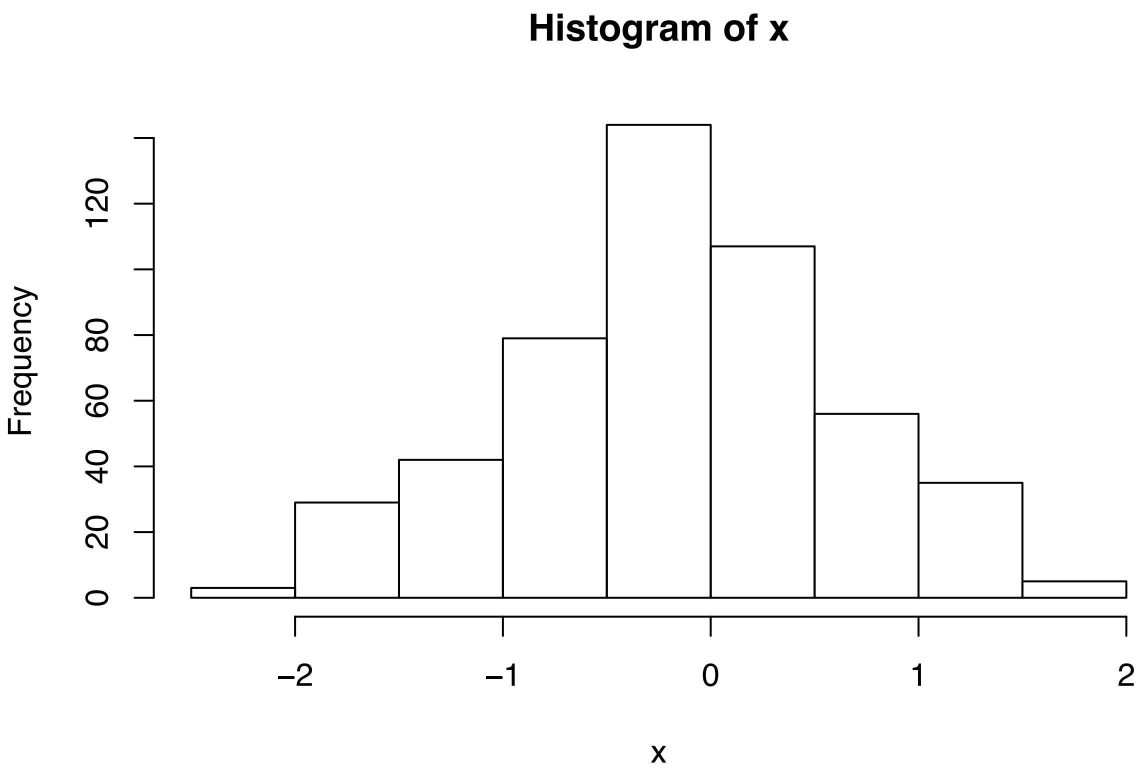
We can see what this looks like but running the iteration many times.

```
delta <- 0.5
N <- 500
x <- numeric(N)
x[1] <- 0
set.seed(2018-06-04)
for(i in 2:N) {
        eps <- runif(1, -delta, delta)
        y <- x[i-1] + eps
        alpha <- min(dnorm(y, log = TRUE) - dnorm(x[i-1], log = TRUE), 0)
        u <- runif(1, 0, 1)
        if(log(u) <= alpha)
                x[i] <- y
        else
                x[i] <- x[i-1]
}
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.1314 -0.6135 -0.1485 -0.1681  0.3034  1.8465
```

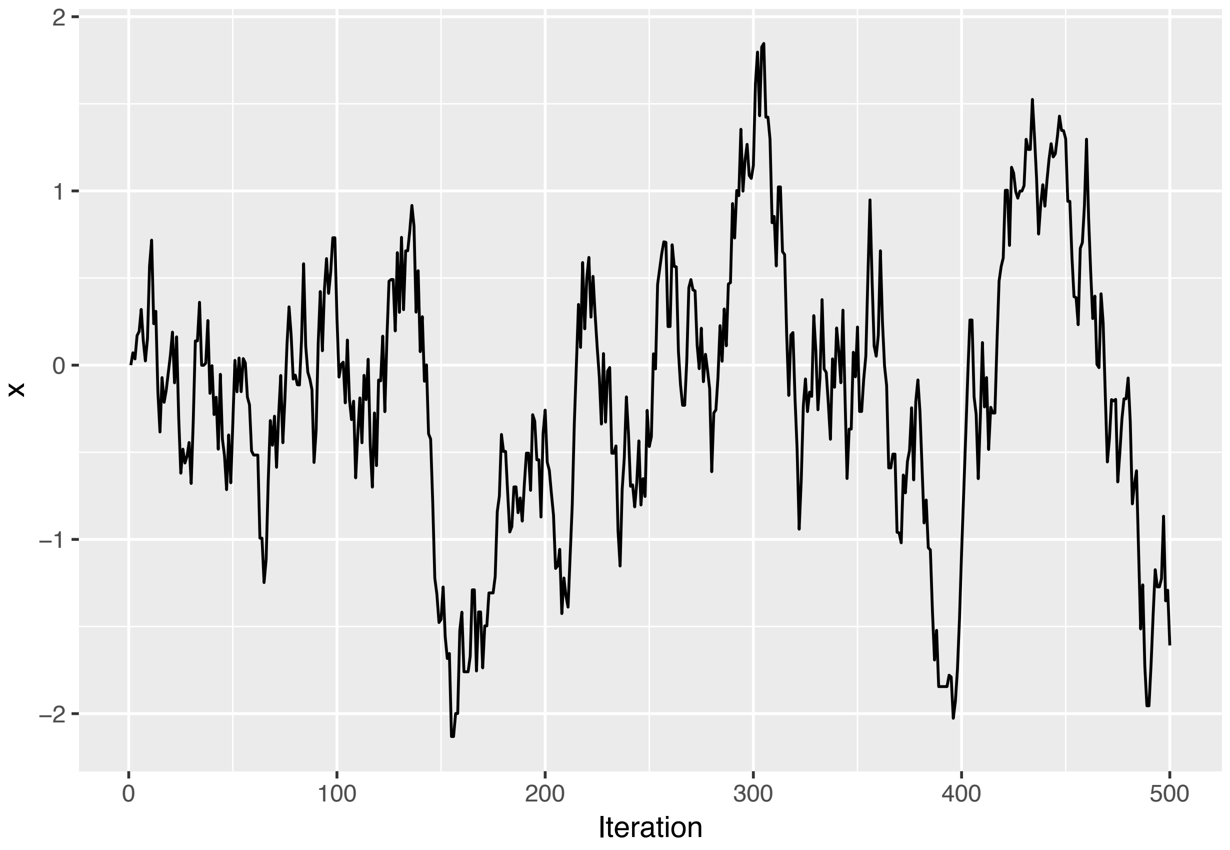
We can take a look at a histogram of the samples to see if they look like a Normal distribution.

```
hist(x)
```



We can also look at a trace plot of the samples to see how the chain moved around.

```
library(ggplot2)
qplot(1:N, x, geom = "line", xlab = "Iteration")
```



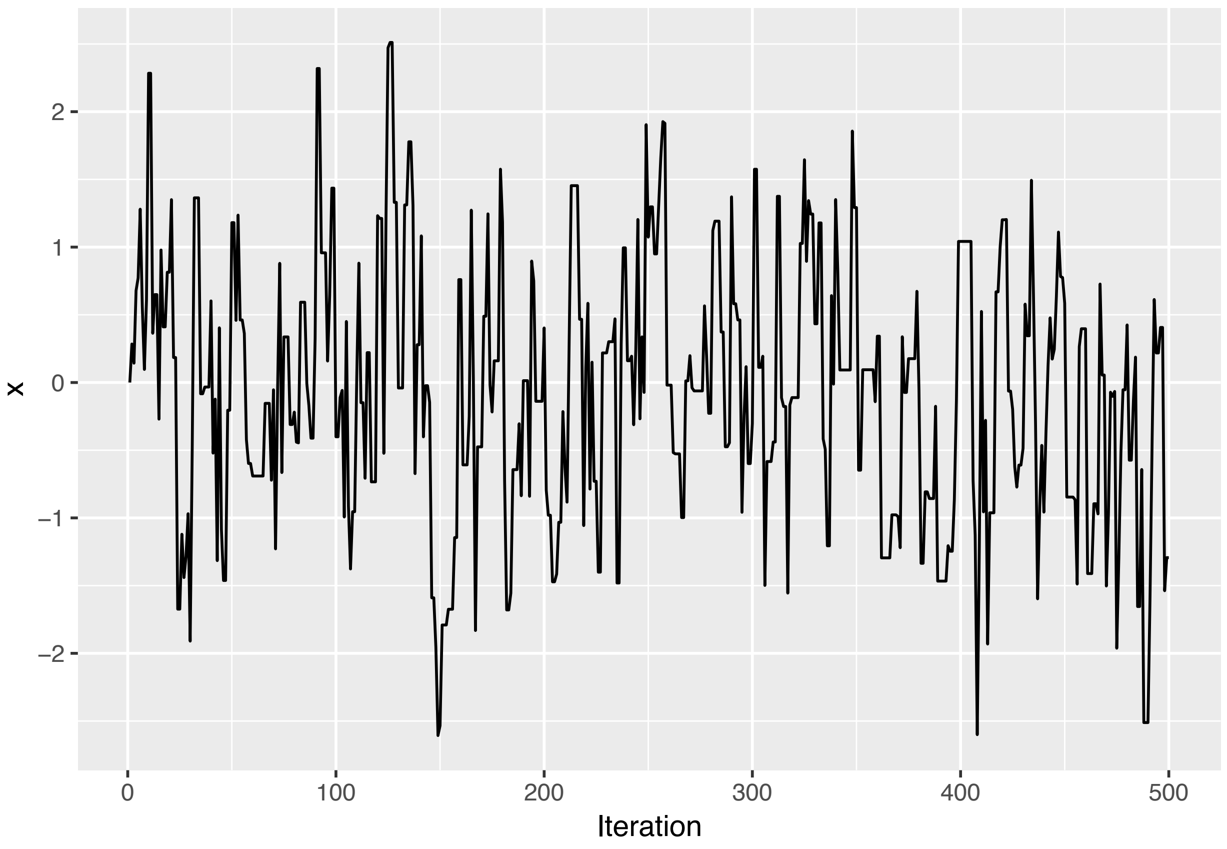
What would happen if we changed the value of δ? Here we make δ bigger and we can see the effect on the trace plot.

```
delta <- 2
N <- 500
x <- numeric(N)
x[1] <- 0
set.seed(2018-06-04)
for(i in 2:N) {
        eps <- runif(1, -delta, delta)
        y <- x[i-1] + eps
        alpha <- min(dnorm(y, log = TRUE) - dnorm(x[i-1], log = TRUE), 0)
        u <- runif(1, 0, 1)
        if(log(u) <= alpha)
                x[i] <- y
        else
                x[i] <- x[i-1]
}
summary(x)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.60714 -0.72944 -0.05603 -0.07395  0.53416  2.51142
```

Now look at the trace plot.

```
library(ggplot2)
qplot(1:N, x, geom = "line", xlab = "Iteration")
```



You can see in this trace plot that there are a number of iterations where there is no movement (so that the candidate is rejected, whereas in the above trace plot with the smaller value of δ there was much more movement, although the steps themselves were smaller. In particular, the first chain exhibited a bit more *autocorrelation*. With a larger δ, we are proposing candidates that are much further from the current state, and that may not comport with the Normal distribution. As a result, the candidates are more likely to be rejected.

This raises an important in Markov chain samplers: tuning. The theory does not depend on a specific value of δ; both chains above should converge to a Normal distribution. However, the speed with which they converge and amount of the sample space that is explored does depend on δ. Hence, the sampler can be tuned to improve its efficiency.

7.2.2 Independence Metropolis Algorithm

The independence Metropolis algorithm defines a transition density as $q(y \mid x) = q(y)$. In other words, the candidate proposals do not depend on the current state x. Otherwise, the algorithm works the same as the original Metropolis-Hastings algorithm, with a modified acceptance ratio,

$$\alpha(y \mid x) = \min\left\{\frac{\pi(y)q(x)}{\pi(x)q(y)}, 1\right\}$$

The independence sampler seems to work well in situations where rejection sampling might be reasonable, i.e. relatively low-dimensional problems. In particular, it has the interesting property that if

$$C = \sup_x \frac{\pi(x)}{q(x)} < \infty$$

then

$$\|\pi_n - \pi\| \leq k\rho^n$$

for $0 < \rho < 1$ and some constant $k > 0$. So if we have the same conditions under which rejection sampling is allowed, then we can prove that convergence of the chain to the stationary distribution is geometric with rate ρ. The value of ρ depends on C; if C is close to 1, then ρ will be small. This is an interesting property of this sampler, but it is largely of theoretical interest.

7.2.3 Slice Sampler

Given the current state x, generate

$$y \sim \text{Unif}(0, \pi(x))$$

Given y, generate a new state $x^\star$

$$x^\star \sim \text{Unif}(\{x : \pi(x) \geq y\})$$

Note that

$$\pi(x) = \int \mathbf{1}\{0 \leq y \leq \pi(x)\}\, dy.$$

Therefore, $f(x, y) = \mathbf{1}\{0 \leq y \leq \pi(x)\}$ is a joint density. The slice sampler method creates an auxiliary variable y and the intergrates it out later to give back our original target density $\pi(x)$.

As depicted in the illustration, the slice sampler can be useful for densities that might have multiple modes where it's easy to get stuck around one model. The sampling of $x^\star$ allows one to jump easily between modes. However, sampling $x^\star$ may be difficult depending on the complexity of $\pi(x)$ and ultimately may not be worth the effort. If $\pi(x)$ is very wiggly or is high-dimensional, then calculating the region $\{x : \pi(x) \geq y\}$ will be very difficult.

7.2.4 Hit and Run Sampler

The hit and run sampler combines ideas from line search optimization methods with MCMC sampling. Here, suppose we have the current state x in p-dimensions and we want to propose a new state. Let e be a random p-dimensional vector that indicates a random direction in which to travel. Then construct the density

$$p(r) \propto \pi(x + re).$$

where r is a scalar (and hence $p(r)$ is a 1-dimensional density). From this density, sample a value for r and set the proposal to be

$$x^\star = x + re.$$

This process has teh advantage that it chooses more directions than a typical Gibbs sampler (see below) and does not require a multi-dimensional proposal distribution. However, sampling from the density $p(r)$ may not be obvious as there is no guaranteed closed form solution.

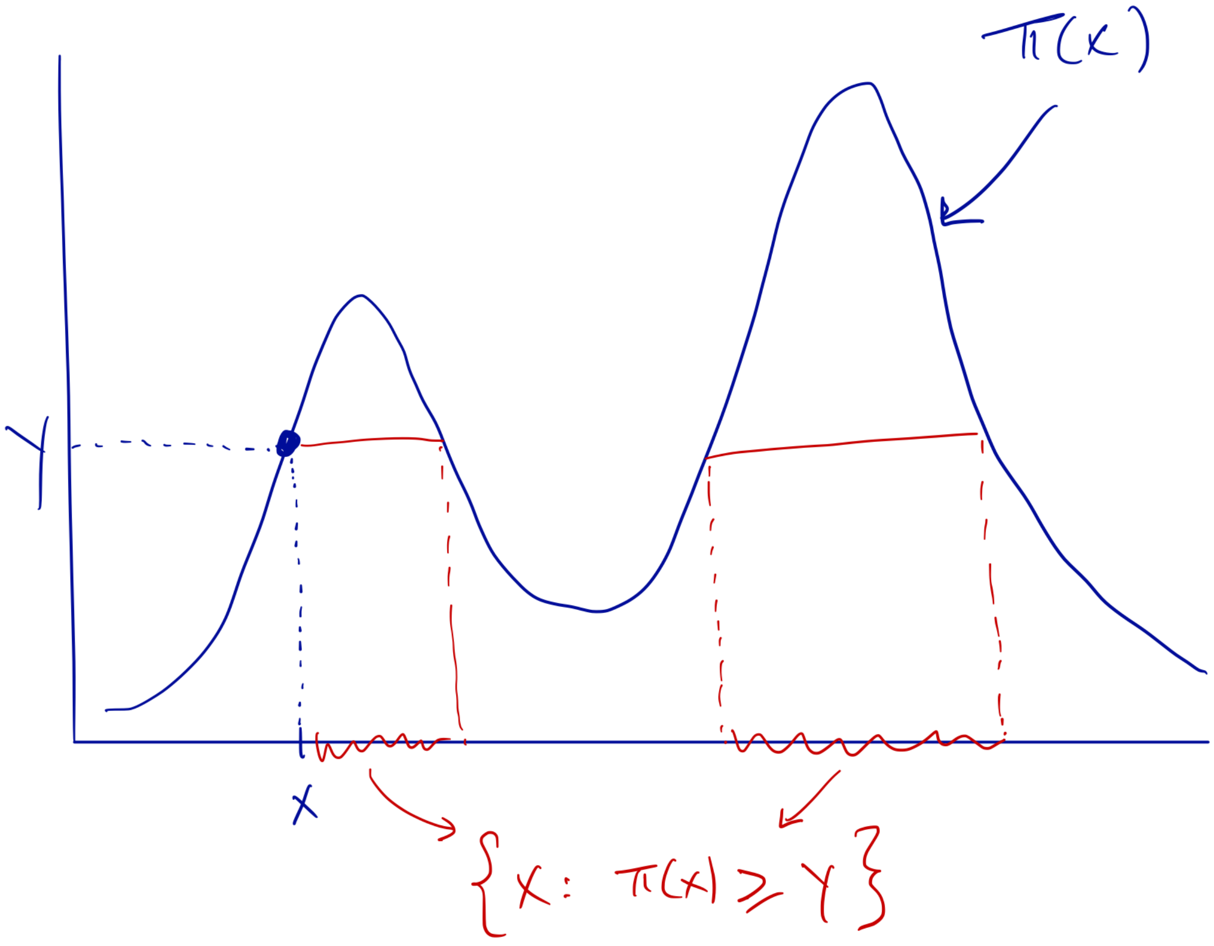


Figure 10: Slice Sampler

7.2.5 Single Component Metropolis-Hastings

The standard Metropolis algorithm updates the entire parameter vector at once with a single step. Hence, a p-dimensional parameter vector must have a p-dimensional proposal distribution q. However, it is sometimes simpler to update individual components of the parameter vector one at a time, in an algortihm known as single component Metropolis-Hastings (SCMH).

Let $x^{(n)} = \left(x_1^{(n)}, x_2^{(n)}, \dots, x_p^{(n)}\right)$ be a p-dimensional vector representing our parameters of interest at iteration n. Define

$$x_{-i} = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_p)$$

as the vector x with the ith component removed. The SCMH algorithm updates each paramater in x one at a time. All that is needed to update the ith component of $x^{(n)}$ is a proposal distribution $q(y \mid x_i^{(n)}, x_{-i}^{(n)})$ for proposing a new value of x_i given the current value of x_i and all the other components.

At iteration n, SCMH updates the ith component via the following steps.

1. Sample $y_i \sim q_i(y \mid x_i^{(n)}, x_{-i}^{(n)})$ as a proposal for component i.
2. Let
$$\alpha(y_i \mid x^{(n)}) = \min\left(\frac{\pi(y_i \mid x_{-i}^{(n)})\, q(x_i \mid y_i, x_{-i}^{(n)})}{\pi(x_i \mid x_{-i}^{(n)})\, q(y_i \mid x_i, x_{-i}^{(n)})}, 1\right)$$
3. Accept y_i for component i with probablity $\alpha(y_i \mid x^{(n)})$.
4. Repeat 1–3 p times until every component is updated.

7.3 Gibbs Sampler

The attraction of an algorithm like single component Metropolis-Hastings is that it converts a p-dimensional problem into p separate 1-dimensional problems, each if which is likely simple to solve. This advantage is not unlike that seen with coordinate descent algorithms discussed previously. SCMH can however be very exploratory and may not be efficient in exploring the parameter space.

Gibbs sampling is a variant of SCMH that uses full conditional distributions for the proposal distribution for each component. Given a target density $\pi(x) = \pi(x_1, \dots, x_p)$, we cycle through sampling from $\pi(x_i \mid x_{-i})$ to update the ith component. Rather than go though an accept/reject step as with Metropolis-Hastings, we always accept, for reasons that will be detailed below.

For example, with a three-component density $\pi(x, y, z)$, the full conditional distributions associated with this density are

$$\pi(x \mid y, z),$$
$$\pi(y \mid x, z),$$

and

$$\pi(z \mid x, y).$$

If our current state is (x_n, y_n, z_n) at the nth iteration, then we update our parameter values with the following steps.

1. Sample $x_{n+1} \sim \pi(x \mid y_n, z_n)$
2. Sample $y_{n+1} \sim \pi(y \mid x_{n+1}, z_n)$
3. Sample $z_{n+1} \sim \pi(z \mid x_{n+1}, y_{n+1})$

At each step of the sampling we use the most recent values of all the other components in the full conditional distribution. In a landmark paper, Geman and Geman showed that if $p(x_n, y_n, z_n)$ is the density of our parameters at the nth iteration, then as $n \to \infty$,

$$\begin{aligned}
p(x_n, y_n, z_n) &\to \pi(x, y, z)\\
p(x_n) &\to \pi(x)\\
p(y_n) &\to \pi(y)\\
p(z_n) &\to \pi(z)
\end{aligned}$$

7.3.1 Example: Bivariate Normal Distribution

Suppose we want to simulate from a bivariate Normal distribution with mean $\mu = (\mu_1, \mu_2)$ and covariance matrix

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \rho \\ \rho & \sigma_2^2 \end{pmatrix}$$

Although there are exact ways to do this, we can make use of Gibbs sampling too simulate a Markov chain that will converge to a bivariate Normal. At the nth iteration with state vector (x_n, y_n), we can update our state with the following steps.

1. Sample $x_{n+1} \sim \mathcal{N}(\mu_1 + \rho\frac{\sigma_1}{\sigma_2}(y_n - \mu_2),\, \sigma_1^2(1-\rho))$
2. Sample $y_{n+1} \sim \mathcal{N}(\mu_2 + \rho\frac{\sigma_2}{\sigma_1}(x_{n+1} - \mu_1),\, \sigma_2^2(1-\rho))$

If ρ is close to 1, then this algorithm will likely take a while to converge. But it is simple to simulate and demonstrates the mechanics of Gibbs sampling.

7.3.2 Example: Normal Likelihood

Suppose we observe $y_1, \dots, y_n \overset{\text{iid}}{\sim} \mathcal{N}(\mu, \tau^{-1})$, where τ^{-1} is the *precision* of the Normal distribution. We can use independent priors for the two parameters as

$$\begin{aligned}
\mu &\sim \mathcal{N}(0, w^{-1})\\
\tau &\sim \text{Gamma}(\alpha, \beta)
\end{aligned}$$

Because the priors are independent here, we will have to use Gibbs sampling to sample from the posterior distribution of μ and τ.

Recall that with Bayes' Theorem, the joint posterior of μ and τ is

$$\begin{aligned}
\pi(\mu, \tau \mid \mathbf{y}) &\propto \pi(\mu, \tau, \mathbf{y})\\
&= f(\mathbf{y} \mid \mu, \tau)p(\mu)p(\tau)
\end{aligned}$$

where $f(\mathbf{y} \mid \mu, \tau)$ is the joint Normal density of the observed data.

In the Gibbs sampling procedure there will be two full conditional distributions that we will want to sample from. They are

$$\begin{aligned}
p(\mu \mid \tau, \mathbf{y}) &\propto f(\mathbf{y} \mid \mu, \tau)p(\mu)p(\tau) \\
&\propto f(\mathbf{y} \mid \mu, \tau)p(\mu) \\
&\propto \exp\left(-\frac{\tau}{2}\sum(y_i - \mu)^2\right)\exp\left(-\frac{w}{2}\mu^2\right) \\
&= \exp\left(-\frac{(n\tau + w)}{2}\mu^2 - \left(\sum y_i\right)\tau\mu\right) \\
&= \mathcal{N}\left(\frac{\tau}{n\tau + w}\sum y_i,\ \frac{1}{n\tau + w}\right)
\end{aligned}$$

and

$$\begin{aligned}
p(\tau \mid \mu, \dagger) &\propto f(\mathbf{y} \mid \mu, \tau)p(\mu)p(\tau) \\
&\propto f(\mathbf{y} \mid \mu, \tau)p(\tau) \\
&\propto \tau^{\frac{n}{2}}\exp\left(-\frac{\tau}{2}\sum(y_i - \mu)^2\right)\tau^{\alpha-1}\exp(-\tau\beta) \\
&= \tau^{\alpha-1+\frac{n}{2}}\exp\left(-\tau\left[\beta + \frac{1}{2}\sum(y_i - \mu)^2\right]\right) \\
&= \text{Gamma}\left(\alpha + \frac{n}{2},\ \beta + \frac{1}{2}\sum(y_i - \mu)^2\right)
\end{aligned}$$

The Gibbs sampler therefore alternates between sampling from a Normal distribution and a Gamma distribution. In this case, the priors were chosen so that the full conditional distributions could be sampled in closed form.

7.3.3 Gibbs Sampling and Metropolis Hastings

Gibbs sampling has a fairly straightforward connection to the single component Metropolis-Hastings algorithm described earlier. We can reframe the Gibbs sampling algorithm for component i at iteration n as

1. Sample $y_i \sim q_i(y \mid x_i^{(n)}, x_{-i}^{(n)}) = \pi(y \mid x_{-i}^{(n)})$. Here, the proposal distribution simply the full conditional distribution of x_i given x_{-i}.
2. The acceptance probability is then

$$\alpha(y_i \mid x_i, x_{-i}) = \min\left(\frac{\pi(y_i \mid x_{-i})\pi(x_i \mid x_{-i})}{\pi(x_i \mid x_{-i})\pi(y_i \mid x_{-i})}, 1\right) = 1$$

The Gibbs sampling algorithm is like the SCMH algorithm but it always accepts the proposal.

7.3.4 Hybrid Gibbs Sampler

Given the relationship between Gibbs sampling and SCMH, we can use this to extend the basic Gibbs algorithm. In some cases, we will not be able to sample directly from the full conditional distribution of a component. In those cases, we can substitute a standard Metropolis-Hastings step with a proposal/acceptance procedure.

For components that require a Metropolis-Hastings step, we will need to come up with a proposal density for those components. In this case, the *target density* is simply the full conditional of component i given the other components. For component i then, the algorithm procedes as

1. Sample $y_i \sim q(y \mid x)$
2. Compute the acceptance probability

$$\alpha(y_i \mid x) = \min\left(\frac{\pi(y_i \mid x_{-i})q(x_i \mid x)}{\pi(x_i \mid x_{-i})q(y_i \mid x)}, 1\right)$$

   We then accept the proposal with probability $\alpha(y_i \mid x)$.

Regardless of whether the proposal is accepted or not, we can move on to the next component of the parameter vector.

7.3.5 Reparametrization

In some Metropolis-Hastings or hybrid Gibbs sampling problems we may have parameters where it is easier to sample from a full conditional of a transformed version of the parameter. For example, we may need to sample from the full conditional $p(\lambda \mid \cdot)$ of a parameter that only takes values between 0 and 1. Doing something like a random walk proposal can be tricky given the restricted range.

One solution is the transform the parameter to a space that has a infinte range. For a parameter $\lambda \in (0, 1)$ we can use the logit transformation to map it to $(-\infty, \infty)$, i.e.

$$z = \text{logit}(\lambda) = \log \frac{\lambda}{1-\lambda}.$$

Then we can do the proposal and acceptance steps on the transformed parameter z, which has an infinite domain. However, we need to be careful that the acceptance ratio is calculated properly to account for the nonlinear transformation of λ. In this case we need to compute the determinant of the Jacobian of the transformation mapping z back to λ.

The algorithm at iteration n in the transformed space would work as follows.

1. Sample $z^\star \sim g(z \mid z_n)$, where $z_n = \text{logit}(\lambda_n)$ and g is the proposal density.
2. Compute

$$\alpha(z^\star \mid z_n) = \min\left(\frac{p(\text{logit}^{-1}(z^\star) \mid \cdot)}{p(\text{logit}^{-1}(z_n) \mid \cdot)} \frac{g(z_n \mid z^\star)}{g(z^\star \mid z_n)} \frac{|J(z^\star)|}{|J(z_n)|}, 1\right)$$

   where $|J(z)|$ is the determinant of the Jacobian of the transformation that maps from $z \mapsto \lambda$, i.e. the function

$$\text{logit}^{-1}(z) = \frac{\exp(z)}{1+\exp(z)}.$$

In R, we can easily compute this Jacobian (and that for any other transformation) with the `deriv()` function.

```r
Jacobian <- local({
        J <- deriv(~ exp(x) / (1 + exp(x)), "x",
                   function.arg = TRUE)
        function(x) {
                val <- J(x)
                drop(attr(val, "gradient"))
        }
})
Jacobian(0)
```

```
##    x
## 0.25
```

Becuase the transformation here is for a single parameter, the determinant is straightforward. However, for a multi-dimensional mapping, we would need an extra step to compute the determinant. The `det()` function can be used for this purpose, but bear in mind that it almost always makes sense to use the `logarithm = TRUE` argument to `det()`.

7.3.6 Example: Bernoulli Random Effects Model

Suppose we observe longitudinal binary data that follow

$$\begin{aligned}
y_{ij} &\sim \text{Bernoulli}(\lambda_{ij})\\
\text{logit}(\lambda_{ij}) &= \alpha_i\\
\alpha_i &\sim \mathcal{N}(\mu, \sigma^2)\\
\mu &\sim \mathcal{N}(0, D)\\
\sigma &\sim \text{InverseGamma}(a, b)
\end{aligned}$$

where $i = 1, \dots, n$ and $j = 1, \dots, G$.

For this example, assume that the hyperparameters a, b, and D are known. The goal is to sample from the posterior of μ and σ. The joint posterior of the parameters given the data is

$$\begin{aligned}
\pi(\mu, \sigma, \alpha \mid \mathbf{y}) \propto & \prod_{i=1}^{n}\left[\prod_{j=1}^{G} p(y_{ij} \mid \text{logit}^{-1}(\alpha_i)\right]\\
& \times\varphi(\alpha_i \mid \mu, \sigma^2)\varphi(\mu \mid 0, D)g(\sigma)
\end{aligned}$$

where $p()$ is the Bernoulli density, φ is the Normal density, and $g()$ is the inverse gamma density.

To implement the Gibbs sampler, we need to cycle through three classes of full conditional distributions. First is the full conditional for σ, which can be written in closed form given the prior. In order to compute the full conditional, we simply select everything from the full posterior that has a σ in it.

$$\begin{aligned}
p(\sigma \mid \cdot) &\propto \left[\prod_{i=1}^{n} \varphi(\alpha_i \mid \mu, \sigma^2)\right] g(\sigma)\\
&= \text{InverseGamma}\left(a + \frac{n}{2}, b + \frac{1}{2}\sum(\alpha_i - \mu)^2\right)
\end{aligned}$$

Similarly, we can pick out all the components of the full posterior that have a μ and the full conditional distribution for μ is

$$\begin{aligned}
p(\mu \mid \cdot) &\propto \left[\prod_{i=1}^{n} \varphi(\alpha_i \mid \mu, \sigma^2)\right] \varphi(\mu \mid, 0, D)\\
&= \mathcal{N}\left(\frac{D}{D + \sigma^2/n}\,\bar{\alpha},\ \frac{\sigma^2/n}{\sigma^2/n + D}D\right)
\end{aligned}$$

where $\bar{\alpha} = \frac{1}{n}\sum \alpha_i$.

Finally, the full conditionals for α_i can be compute independently because the α_is are assumed to be independent.

$$p(\alpha_i \mid \cdot) \propto \left[\prod_{j=1}^{G} p(y_{ij} \mid \text{logit}^{-1}(\alpha_i))\right] \varphi(\alpha_i \mid \mu, \sigma^2).$$

Unfortunately, there is no way to simplify this further with the combination of the Bernoulli likelihood and the Normal random effects distribution. Therefore, some proposal/acceptance step will have to be run to sample from the α_is.

7.4 Monitoring Convergence

Once you've got your simulated Markov chain running, the question arises regarding when it should be stopped. On an infinite timeline, we know from theory that the samples drawn from the chain will come from the target density. On every *other* timeline, we must design some rule for stopping the chain in order to get things like parameter estimates or posterior distributions.

Determining how and when to stop a chain depends a bit on what exactly you are trying to do. In many cases, regardless of your inferential philosophy, you are trying to obtain an estimate of a parameter in a model. Typically, we estimate these parameters by using a summary statistic like the mean of the posterior distribution. We calculate this summary statistic by drawing samples from the posterior and computing the arithmetic mean.

Hence, we are *estimating a parameter* and in doing so we must worry about the usual things we worry about when estimating a parameter. In particular, we must worry about the uncertainty introduced by our procedure, which in this case is the Markov chain Monte Carlo sampling procedure. If our chain ran to infinity, we would have *no uncertainty* due to MCMC sampling (we would still have other kinds of uncertainty, such as from our finite dataset). But because our chain will not run to infinity, stopping the chain can be thought of as answering the question of how much Monte Carlo uncertainty is acceptable.

7.4.1 Monte Carlo Standard Errors

One way to measure the amount of uncertainty introduced through MCMC sampling is with Monte Carlo standard errors. The thought experiment here is, if we were to repeatedly run our MCMC sampler (with different random number generator seeds!) for a fixed N number of iterations, how much variability would we expect to see in our estimate of the parameter? Monte Carlo standard errors should give us a sense of this variability. If the standard errors are two big, we can run the sampler for longer. Exactly how long we will need to run the sampler to achieve a given standard error will depend on the efficiency and mixing of the sampler.

One way to compute Monte Carlo standard errors is with the method of *batch means.* The idea here is we divide our long MCMC sampler chain into equal size segments and compute our summary statistic on each segment. If the segments are of the proper size, we can think of them as "independent replicates", even though individual samples of the MCMC sampler will not be independent of each other in general. From these replicates, we compute an estimate of variability from the given chain.

More specifically, suppose $x_1, x_2, x_3, \ldots$ are samples from an MCMC sampler that we have run for N iterations and we want to estimate $\mathbb{E}[h(X)]$ where the expectation is take with respect to some posterior distribution. We first decide on a batch size K and let's assume that $N/K = M$ where M is an integer. We then divide the chain into segments $x_1, \ldots, x_K$, $x_{K+1}, \ldots, x_{2K}$, etc. From here, we can compute our segment summary statistics,

$$
\begin{aligned}
b_1 &= \frac{1}{K}\sum_{i=1}^{K} h(x_i)\\
b_2 &= \frac{1}{K}\sum_{i=K+1}^{2K} h(x_i)\\
&\vdots\\
b_M &= \frac{1}{K}\sum_{i=(M-1)K+1}^{MK} h(x_i)
\end{aligned}
$$

Here, we can confirm that

$$
\bar{b} = \frac{1}{M}\sum_{i=1}^{M} b_i = \frac{1}{KM}\sum_{i=1}^{KM} h(x_i) \longrightarrow \mathbb{E}[h(X)]
$$

as $N \to \infty$.

Once we have the batch means, we can compute a standard error based on the assumed ergodicity of the chain, which gives us,

$$
\sqrt{M}\left(\frac{\bar{b} - \mathbb{E}[h(X)]}{s}\right) \longrightarrow \mathcal{N}(0, 1).
$$

The batch means standard error is the square root of

$$
s^2 = \frac{K}{M}\sum_{i=1}^{M}(b_i - \bar{b})^2.
$$

For a specific implementation of the batch means procedure, one can use the method of Jones, Haran, Caffo, and Neath. The source code for this procedure is available at Murali Haran's web site. Jones, *et al* recommend a batch size of $N^{1/2}$ (or the smallest integer closest to that) and that is the default setting for their software. If the chain is mixing relatively well, a batch size of $N^{1/3}$ can be used and may be more efficient.

It's worth noting that the Monte Carlo standard error is a quantity with units attached to it. Therefore, determining when the standard error is "small enough" will require a certain understanding of the context in which the problem is being addressed. No universal recommendation can be made on how small is small enough. Some view this property as a downside, preferring something that is "unit free", but I see it as an important reminder that these parameters that we are estimating come from the real world and are addressing real problems.

7.4.2 Gelman-Rubin Statistic

Another approach to monitoring the convergence of a MCMC sampler is to think about what we might expect when a chain has "converged". If we were to start multiple parallel chains in many different starting values, the theory claims that they should all eventually converge to the stationary distribution. So after some amount of time, it should be impossible to distinguish between the multiple chains. They should all "look" like the stationary distribution. One way to assess this is to compare the variation between chains to the variation within the chains. If all the chains are "the same", then the between chain variation should be close to zero.

Let $x_1^{(j)}, x_2^{(j)}, \dots$ be samples from the jth Markov chain and suppose there are J chains run in parallel with different starting values.

1. For each chain, first discard D values as "burn-in" and keep the remaining L values, $x_D^{(j)}, x_{D+1}^{(j)}, \ldots, x_{D+L-1}^{(j)}$. For example, you might set $D = L$.
2. Calculate

$$\begin{aligned}
\bar{x}_j &= \frac{1}{L}\sum_{t=1}^{L} x_t^{(j)} \qquad \text{(chain mean)} \\
\bar{x}_{\cdot} &= \frac{1}{J}\sum_{j=1}^{J} \bar{x}_j \qquad \text{(grand mean)} \\
B &= \frac{L}{J=1}\sum_{j=1}^{J}(\bar{x}_j - \bar{x}_{\cdot})^2 \qquad \text{(between chain variance)} \\
s_j^2 &= \frac{1}{L-1}\sum_{t=1}^{L}(x_t^{(j)} - \bar{x}_j)^2 \qquad \text{(within chain variance)} \\
W &= \frac{1}{J}\sum_{j=1}^{J} s_j^2
\end{aligned}$$

3. The Gelman-Rubin statistic is then

$$R = \frac{\frac{L-1}{L}W + \frac{1}{L}B}{W}$$

We can see that as $L \to \infty$ and as $B \downarrow 0$, R approaches the value of 1. One can then reason that we should run our chains until the value of R is close to 1, say in the neighborhood of 1.1 or 1.2.

The Gelman-Rubin statistic is a ratio, and hence unit free, making it a simple summary for any MCMC sampler. In addition, it can be implemented without first specifying a parameter that is to be estimated, unlike Monte Carlo standard errors. Therefore, it can be a useful tool for monitoring a chain before any specific decisions about what kinds of inferences will be made from the model.

While the Gelman-Rubin statistic is intuitively appealing and is practically useful, it can demonstrate some undesirable properties. In particular, Flegal *et al.* report that the Gelman-Rubin statistic can be unstable, leading to both very long runs and very short runs. The result is that the uncertainty of any parameter being estimated with a MCMC sampler will be greater than that estimated using MC standard errors.

Therefore, for the purposes of parameter estimation, it would seem for now that Monte Carlo standard errors are a better way to go. The other benefit of the Monte Carlo standard error approach is that you don't have to worry about burn-in (i.e. no samples are discarded). However, for more general Markov chain monitoring, the Gelman-Rubin statistic may be a reasonable tool to use.

7.5 Simulated Annealing

Simulated annealing is a technique for minimizing functions that makes use of the ideas from Markov chain Monte Carlo samplers, which is why it is in this section of the book. It is a particularly useful method for functions that are very misbehaved and wiggly; the kinds of functions that Newton-style optimizers tend to be bad at minimizing.

Suppose we want to find the global minimum of a function $h(\theta)$, where θ is a vector of parameters in a space S. Ideally, we would simulate a Markov chain whose target density $\pi(\theta)$ was a point mass on the global minimum. This would be great if we could do it! But then we wouldn't have a problem. The idea with simulated annealing is that we build successive approximations to $\pi(\theta)$ until we have an approximation that is very close to the target density.

Let $S^\star = \{\theta \in S : h(\theta) = \min_\theta h(\theta)\}$. Then define $\pi(\theta) \propto 1$ for all $\theta \in S^\star$ and $\pi(\theta) = 0$ for all $\theta \notin S^\star$. In other words, $\pi(\theta)$ is the uniform distribution over all the global minimizers. The ultimate goal is to find some way to sample from $\pi(\theta)$.

We will begin by building an approximate density called $\pi_T(\theta)$ where

$$\pi_T(\theta) \propto \exp(-h(\theta)/T).$$

and where T is called the "temperature". This density as two key properties:

1. As $T \to \infty$ it $\pi_T(\theta)$ approaches the uniform density;
2. As $T \downarrow 0$, $\pi_T(\theta) \to \pi(\theta)$.

The aim is then to draw many samples from $\pi_T(\theta)$, initially with a large value of T, and to lower T towards 0 slowly. As we lower T, the density $\pi_T(\theta)$ will become more and more concentrated around the minima of $h(\theta)$.

Given that $\pi_T(\theta) \to \pi(\theta)$ as $T \downarrow 0$, why not just start with T really small? The problem is that if T is small from the start, then the sampler will quickly get "stuck" in a whatever local model is near our initial value. Once there, it will not be able to jump out and go to an other mode. So the general strategy is to start with a large T so that the sample space can be adequately explored and so we don't get stuck in a local minimum. Then, as we lower the temperature, we can be sure that we have explored as much of the space as feasible.

The sampling procedure is then to first choose a symmetric proposal density $q(\cdot \mid \theta)$. Then, if we are at iteration n with state θ_n,

1. Sample $\theta^\star \sim q(\theta \mid \theta_n)$.
2. Sample $U \sim \text{Unif}(0, 1)$.
3. Compute

$$\begin{aligned}\alpha(\theta^\star \mid \theta_n) &= \min\left(\frac{\exp(-h(\theta^\star)/T)}{\exp(-h(\theta_n)/T)}, 1\right)\\ &= \min(\exp(-(h(\theta^\star) - h(\theta_n))/T), 1)\end{aligned}$$

4. Accept $\theta^\star$ as the next state if $U \le \alpha(\theta^\star \mid \theta_n)$.
5. Decrease T.

Note that if $h(\theta^\star) \le h(\theta_n)$, then we always accept the proposal, so we always go "downhill". If $h(\theta^\star) > h(\theta_n)$ then we accept $\theta^\star$ with some probability < 1. So there is a chance that we go "uphill", which allows us to not get stuck in local minima. As T approaches 0, it becomes much less likely that we will accept an uphill proposal.

If we have that

1. The temperature is decreasing, so that $T_{n+1} < T_n$ for each iteration n;
2. $T_n \downarrow 0$ as $n \to \infty$;
3. T_n decreases according to an appropriate "cooling schedule";

then $\|\pi_{T_n} - \pi\| \longrightarrow 0$.

For the theory to work and for us to be guaranteed to find the global minimizer(s), we need to have a "cooling schedule" for T that is along the lines of

$$T_n = \frac{a}{\log(n + b)}$$

for some $a, b > 0$. Because of the logarithm in the denominator, this cooling schedule is excruciatingly slow, making the simulated annealing algorithm a very slow algorithm to converge.