Unit 11: Optimization

November 14, 2018

References:

- Gentle: *Computational Statistics*
- Lange: *Optimization*
- Monahan: *Numerical Methods of Statistics*
- Givens and Hoeting: *Computational Statistics*
- Materials online from Stanford's EE364a course on convex optimization, including Boyd and Vandenberghe's (online) book Convex Optimization, which is also linked to from the course webpage.

1 Notation

We'll make use of the first derivative (the gradient) and second derivative (the Hessian) of functions. We'll generally denote univariate and multivariate functions (without distinguishing between them) as $f(x)$ with $x = (x_1, \ldots, x_p)$. The (column) vector of first partial derivatives (the gradient) is $f'(x) = \nabla f(x) = (\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_p})^\top$ and the matrix of second partial derivatives (the Hessian) is

$$f''(x) = \nabla^2 f(x) = H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_p} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_p} \\ \vdots & \vdots & \ddots & \\ \frac{\partial^2 f}{\partial x_1 \partial x_p} & \frac{\partial^2 f}{\partial x_2 \partial x_p} & \cdots & \frac{\partial^2 f}{\partial x_p^2} \end{pmatrix}.$$

In considering iterative algorithms, I'll use x_0, $x_1, \ldots, x_t$, x_{t+1} to indicate the sequence of values as we search for the optimum, denoted x^*. x_0 is the starting point, which we must choose (often

carefully). If it's unclear at any point whether I mean a value of x in the sequence or a sub-element of the x vector, let me know, but hopefully it will be clear from context most of the time.

I'll try to use x (or if we're talking explicitly about a likelihood, θ) to indicate the argument with respect to which we're optimizing and Y to indicate data involved in a likelihood. I'll try to use z to indicate covariates/regressors so there's no confusion with x.

2 Overview

The basic goal here is to optimize a function numerically when we cannot find the maximum (or minimum) analytically. Some examples:

1. Finding the MLE for a GLM
2. Finding least squares estimates for a nonlinear regression model,

$$Y_i \sim \mathcal{N}(g(z_i; \beta), \sigma^2)$$

   where $g(\cdot)$ is nonlinear and we seek to find the value of $\theta = (\beta, \sigma^2)$ that best fits the data.

3. Maximizing a likelihood under constraints
4. Fitting a machine learning prediction method

Maximum likelihood estimation and variants thereof is a standard situation in which optimization comes up.

We'll focus on **minimization**, since any maximization of f can be treated as minimization of $-f$. The basic setup is to find the *argument*, x, that minimizes $f(x)$:

$$\arg\min_{x \in D} f(x)$$

where D is the domain. Sometimes $D = \Re^p$ but other times it imposes constraints on x. When there are no constraints, this is unconstrained optimization, where any x for which $f(x)$ is defined is a possible solution. We'll assume that f is continuous as there's little that can be done systematically if we're dealing with a discontinuous function.

In one dimension, minimization is the same as root-finding with the derivative function, since the minimum of a differentiable function can only occur at a point at which the derivative is zero. So with differentiable functions we'll seek to find x s.t. $f'(x) = \nabla f(x) = 0$. To ensure a minimum, we want that for all y in a neighborhood of x^*, $f(y) \geq f(x^*)$, or (for twice differentiable functions) $f''(x^*) = \nabla^2 f(x^*) = H_f(x^*) \geq 0$, i.e., that the Hessian is positive semi-definite.

Different strategies are used depending on whether D is discrete and countable, or continuous, dense and uncountable. We'll concentrate on the continuous case but the discrete case can arise in statistics, such as in doing variable selection.

In general we rely on the fact that we can evaluate f. Often we make use of analytic or numerical derivatives of f as well.

To some degree, optimization is a solved problem, with good software implementations, so it raises the question of how much to discuss in this class. The basic motivation for going into some of the basic classes of optimization strategies is that the function being optimized changes with each problem and can be tricky to optimize, and I want you to know something about how to choose a good approach when you find yourself with a problem requiring optimization. Finding global, as opposed to local, minima can also be an issue.

Note that I'm not going to cover MCMC (Markov chain Monte Carlo) methods, which are used for approximating integrals and sampling from posterior distributions in a Bayesian context and in a variety of ways for optimization. If you take a Bayesian course you'll cover this in detail, and if you don't do Bayesian work, you probably won't have much need for MCMC, though it comes up in MCEM (Monte Carlo EM) and simulated annealing, among other places.

Goals for the unit Optimization is a big topic. Here's what I would like you to get out of this:

1. an understanding of line searches (one-dimensional optimization),
2. an understanding of multivariate derivative-based optimization and how line searches are useful within this,
3. an understanding of derivative-free methods,
4. an understanding of the methods used in R's optimization routines, their strengths and weaknesses, and various tricks for doing better optimization in R, and
5. a basic idea of what convex optimization is and when you might want to go learn more about it.

3 Univariate function optimization

We'll start with some strategies for univariate functions. These can be useful later on in dealing with multivariate functions.

3.1 Golden section search

This strategy requires only that the function be unimodal.

Assume we have a single minimum, in $[a, b]$. We choose two points in the interval and evaluate them, $f(x_1)$ and $f(x_2)$. If $f(x_1) < f(x_2)$ then the minimum must be in $[a, x_2]$, and if the converse in $[x_1, b]$. We proceed by choosing a new point in the new, smaller interval and iterate. At each step we reduce the length of the interval in which the minimum must lie. The primary question involves what is an efficient rule to use to choose the new point at each iteration.

Suppose we start with x_1 and x_2 s.t. they divide $[a, b]$ into three equal segments. Then we use $f(x_1)$ and $f(x_2)$ to rule out either the leftmost or rightmost segment based on whether $f(x_1) < f(x_2)$. If we have divided equally, we cannot place the next point very efficiently because either x_1 or x_2 equally divides the remaining space, so we are forced to divide the remaining space into relative lengths of 0.25, 0.25, and 0.5. The next time around, we may only rule out the shorter segment, which leads to inefficiency.

The efficient strategy is to maintain the *golden ratio* between the distances between the points using $\phi = (\sqrt{5} - 1)/2 \approx .618$, the golden ratio. We start with $x_1 = a + (1 - \phi)(b - a)$ and $x_2 = a + \phi(b - a)$. Then suppose $f(x_1) < f(x_2)$. We now choose to place x_3 s.t. it uses the golden ratio in the interval $[a, x_1]$: $x_3 = a + (1 - \phi)(x_2 - a)$. Because of the way we've set it up, we once again have the third subinterval, $[x_1, x_2]$, of equal length as the first subinterval, $[a, x_3]$. The careful choice allows us to narrow the search interval by an equal proportion,$1 - \phi$, in each iteration. Eventually we have narrowed the minimum to between x_{t-1} and x_t, where the difference $|x_t - x_{t-1}|$ is sufficiently small (within some tolerance - see Section 4 for details), and we report $(x_t + x_{t-1})/2$. We'll see an example of this on the board in class.

3.2 Bisection method

The bisection method requires the existence of the first derivative but has the advantage over the golden section search of halving the interval at each step. We again assume unimodality.

We start with an initial interval (a_0, b_0) and proceed to shrink the interval. Let's choose a_0 and b_0, and set x_0 to be the mean of these endpoints. Now we update according to the following algorithm, assuming our current interval is $[a_t, b_t]$:

$$[a_{t+1}, b_{t+1}] = \begin{cases} [a_t, x_t] & \text{if} f'(a_t)f'(x_t) < 0 \\ [x_t, b_t] & \text{if} f'(a_t)f'(x_t) > 0 \end{cases}$$

and set x_{t+1} to the mean of a_{t+1} and b_{t+1}. The basic idea is that if the derivative at both a_t and x_t is negative, then the minimum must be between x_t and b_t, based on the intermediate value theorem.

If the derivatives at a_t and x_t are of different signs, then the minimum must be between a_t and x_t.

Since the bisection method reduces the size of the search space by one-half at each iteration, one can work out that each decimal place of precision requires 3-4 iterations. Obviously bisection is more efficient than the golden section search because we reduce by $0.5 > 0.382 = 1 - \phi$, so we've gained information by using the derivative. It requires an evaluation of the derivative however, while golden section just requires an evaluation of the original function.

Bisection is an example of a *bracketing* method, in which we trap the minimum within a nested sequence of intervals of decreasing length. These tend to be slow, but if the first derivative is continuous, they are robust and don't require that a second derivative exist.

3.3 Newton-Raphson (Newton's method)

3.3.1 Overview

We'll talk about Newton-Raphson (N-R) as an optimization method rather than a root-finding method, but they're just different perspectives on the same algorithm.

For N-R, we need two continuous derivatives that we can evaluate. The benefit is speed, relative to bracketing methods. We again assume the function is unimodal. The minimum must occur at x^* s.t. $f'(x^*) = 0$, provided the second derivative is non-negative at x^*. So we aim to find a zero (a root) of the first derivative function. Assuming that we have an initial value x_0 that is close to x^*, we have the Taylor series approximation

$$f'(x) \approx f'(x_0) + (x - x_0)f''(x_0).$$

Now set $f'(x) = 0$, since that is the condition we desire (the condition that holds when we are at x^*), and solve for x to get

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)},$$

and iterate, giving us updates of the form $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$. What are we doing intuitively? Basically we are taking the tangent to $f(x)$ at x_0 and extrapolating along that line to where it crosses the x-axis to find x_1. We then reevaluate $f(x_1)$ and continue to travel along the tangents.

One can prove that if $f'(x)$ is twice continuously differentiable, is convex, and has a root, then N-R converges from any starting point.

Note that we can also interpret the N-R update as finding the analytic minimum of the quadratic Taylor series approximation to $f(x)$.

Newton's method converges very quickly (as we'll discuss in Section 4), but if you start too far from the minimum, you can run into serious problems.

3.3.2 Secant method variation on N-R

Suppose we don't want to calculate the second derivative required in the divisor of N-R. We might replace the analytic derivative with a discrete difference approximation based on the secant line joining $(x_t, f'(x_t))$ and $(x_{t-1}, f'(x_{t-1}))$, giving an approximate second derivative:

$$f''(x_t) \approx \frac{f'(x_t) - f'(x_{t-1})}{x_t - x_{t-1}}.$$

For this variant on N-R, we need two starting points, x_0 and x_1.

An alternative to the secant-based approximation is to use a standard discrete approximation of the derivative such as

$$f''(x_t) \approx \frac{f'(x_t + h) - f'(x_t - h)}{2h}.$$

3.3.3 How can Newton's method go wrong?

Let's think about what can go wrong - namely when we could have $f(x_{t+1}) > f(x_t)$? Basically, if $f'(x_t)$ is relatively flat, we can get that $|x_{t+1} - x^*| > |x_t - x^*|$. We'll see an example on the board and the demo code (see below). Newton's method can also go uphill when the second derivative is negative, with the method searching for a maximum.

First let's see an example of divergence.

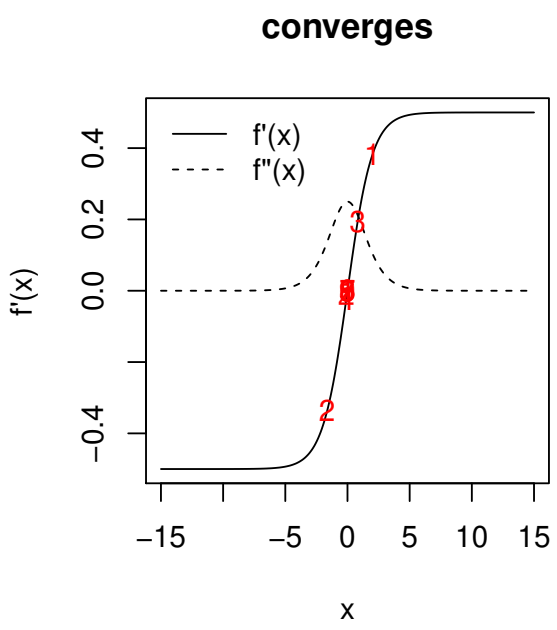
```
par(mfrow = c(1,2))
fp <- function(x, theta = 1){
        exp(x*theta)/(1+exp(x*theta)) - .5
}
fpp <- function(x, theta = 1){
        exp(x*theta)/((1+exp(x*theta))^2)
}
xs <- seq(-15, 15, len = 300)

## good starting point
x0 <- 2
xvals <- c(x0,rep(NA,9))
for(t in 2:10){
        xvals[t]=xvals[t-1] - fp(xvals[t-1]) / fpp(xvals[t-1])
}
print(xvals)
```

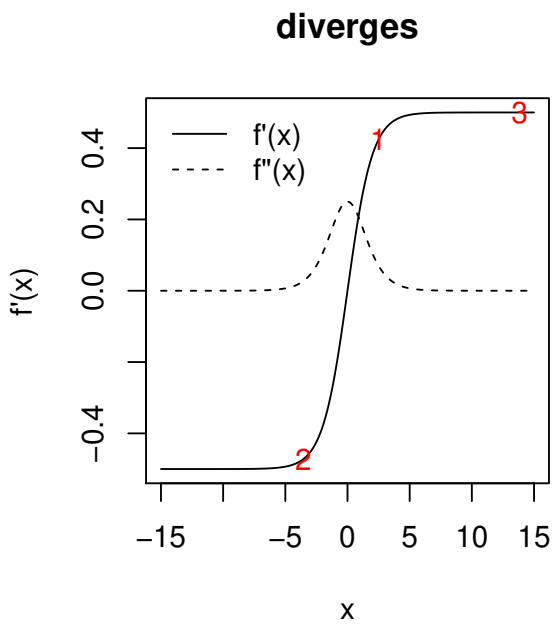
```
##  [1]  2.000000e+00 -1.626860e+00  8.188046e-01 -9.460983e-02  1.412056e-
##  [6] -4.691188e-13  6.142005e-17  6.142005e-17  6.142005e-17  6.142005e-
```

```
plot(xs, fp(xs), type = 'l', xlab = 'x', ylab = "f'(x)", main ='converges')
lines(xs, fpp(xs), lty = 2)
legend('topleft', bty = 'n', lty = c(1,2), legend = c("f'(x)", 'f"(x)'))
points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = 'red')

## bad starting point
x0 <- 2.5
xvals <- c(x0,rep(NA,9))
for(t in 2:10){
        xvals[t]=xvals[t-1] - fp(xvals[t-1]) / fpp(xvals[t-1])
}
print(xvals)
```

```
##  [1]  2.500000e+00 -3.550204e+00  1.384565e+01 -5.152876e+05            I
##  [6]           NaN           NaN           NaN           NaN           Na
```

```
plot(xs, fp(xs), type = 'l', xlab = 'x', ylab = "f'(x)", main = 'diverges')
lines(xs, fpp(xs), lty = 2)
legend('topleft', lty = c(1,2), legend = c("f'(x)", 'f"(x)'), bty = 'n')
points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = 'red')
```






```
## whoops
```

Now let's see an example of climbing uphill and finding a local maximum rather than minimum.

```
## example of mistakenly climbing uphill

par(mfrow = c(3,1))

# original fxn
f <- function(x) cos(x)
# gradient
fp <- function(x) -sin(x)
# second derivative
fpp <- function(x) -cos(x)
xs <- seq(0, 2*pi, len = 300)

x0 <- 1 # starting point
fp(x0) # negative

## [1] -0.841471

fpp(x0) # negative
```

```
## [1] -0.5403023

x1 <- x0 - fp(x0)/fpp(x0) # whoops, we've gone uphill
## because of the negative second derivative
xvals <- c(x0,rep(NA,9))
for(t in 2:10){
        xvals[t]=xvals[t-1]-fp(xvals[t-1])/fpp(xvals[t-1])
}
xvals

##  [1]  1.000000e+00 -5.574077e-01  6.593645e-02 -9.572192e-05  2.923566e-
##  [6]  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+

plot(xs, f(xs), type = 'l', xlab = 'x', ylab = "f'(x)", lwd = 2,
     main = 'uphill to local maximum',)
lines(xs, fp(xs))
lines(xs, fpp(xs), lty = 2)
legend('bottomright', lty = c(1,1,2), lwd = c(2, 1, 1),
       legend = c("f(x)", "f'(x)", 'f"(x)'), bty = 'n')
points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = 'red')
## and we've found a maximum rather than a minimum...

## in contrast, with better starting points we can find the minimum

x0 <- 2 # ok starting point
fp(x0)

## [1] -0.9092974

fpp(x0)

## [1] 0.4161468

x1 <- x0 - fp(x0)/fpp(x0)
xvals <- c(x0,rep(NA,9))
for(t in 2:10){
        xvals[t]=xvals[t-1]-fp(xvals[t-1])/fpp(xvals[t-1])
}
xvals
```

```
##  [1] 2.000000 4.185040 2.467894 3.266186 3.140944 3.141593 3.141593
##  [8] 3.141593 3.141593 3.141593
```

```
# oscillates and comes close to diverging but converges
plot(xs, f(xs), type = 'l', xlab = 'x', ylab = "f'(x)", lwd = 2,
     main = 'converges to minimum, nearly diverges')
lines(xs, fp(xs))
lines(xs, fpp(xs), lty = 2)
legend('bottomright', lty = c(1,1,2), lwd = c(2, 1, 1),
       legend = c("f(x)", "f'(x)", 'f"(x)'), bty = 'n')
points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = 'red')
```

```
x0 <- 2.5 # good starting point
fp(x0)
```

```
## [1] -0.5984721
```

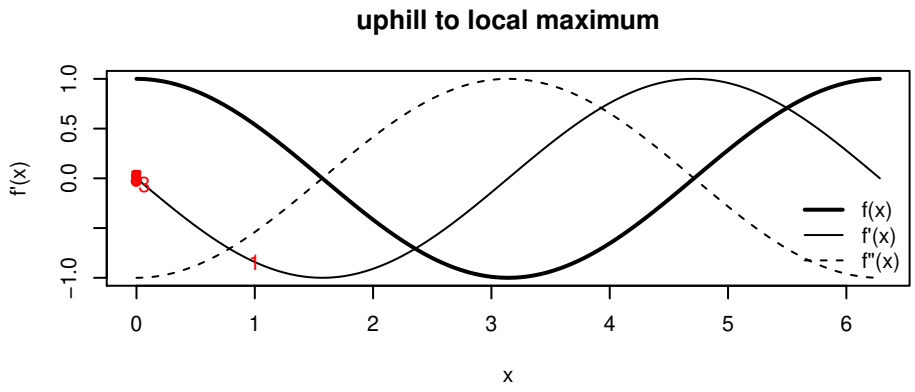
```
fpp(x0)
```

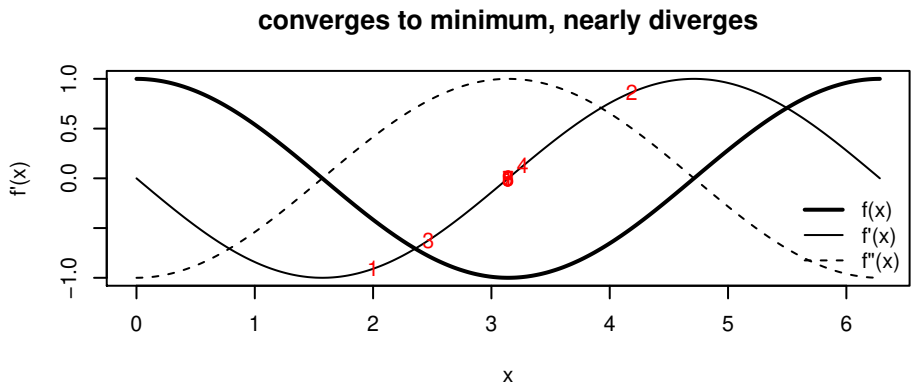
```
## [1] 0.8011436
```

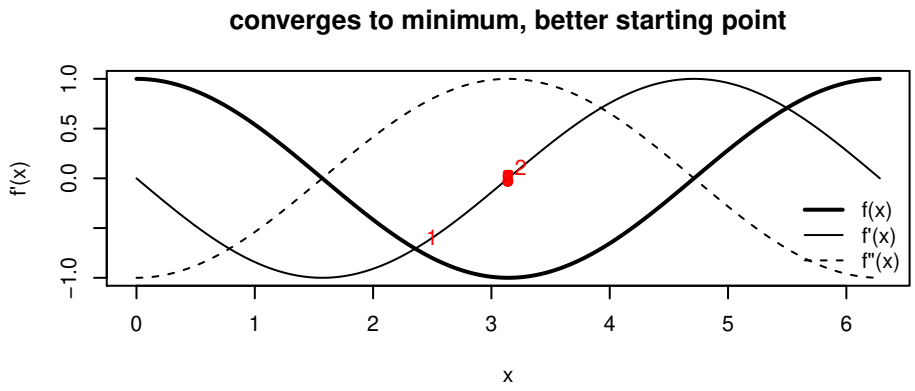
```
x1 <- x0 - fp(x0)/fpp(x0)
xvals <- c(x0,rep(NA,9))
for(t in 2:10){
        xvals[t]=xvals[t-1]-fp(xvals[t-1])/fpp(xvals[t-1])
}
xvals
```

```
##  [1] 2.500000 3.247022 3.141200 3.141593 3.141593 3.141593 3.141593
##  [8] 3.141593 3.141593 3.141593
```

```
## converges quickly
plot(xs, f(xs), type = 'l', xlab = 'x', ylab = "f'(x)", lwd = 2,
     main = 'converges to minimum, better starting point')
lines(xs, fp(xs))
lines(xs, fpp(xs), lty = 2)
legend('bottomright', lty = c(1,1,2), lwd = c(2, 1, 1),
       legend = c("f(x)", "f'(x)", 'f"(x)'), bty = 'n')
points(xvals, fp(xvals), pch = as.character(1:length(xvals)), col = 'red')
```







One nice, general idea is to use a fast method such as Newton's method *safeguarded* by a robust, but slower method. Here's how one can do this for N-R, safeguarding with a bracketing method such as bisection. Basically, we check the N-R proposed move to see if N-R is proposing a step outside of where the root is known to lie based on the previous steps and the gradient values for those steps. If so, we could choose the next step based on bisection.

Another approach is backtracking. If a new value is proposed that yields a larger value of the function, backtrack to find a value that reduces the function. One possibility is a line search but given that we're trying to reduce computation, a full line search is often unwise computationally

(also in the multivariate Newton's method, we are in the middle of an iterative algorithm for which we will just be going off in another direction anyway at the next iteration). A basic approach is to keep backtracking in halves. A nice alternative is to fit a polynomial to the known information about that slice of the function, namely $f(x_{t+1})$, $f(x_t)$, $f'(x_t)$ and $f''(x_t)$ and find the minimum of the polynomial approximation.

4 Convergence ideas

4.1 Convergence metrics

We might choose to assess whether $f'(x_t)$ is near zero, which should assure that we have reached the critical point. However, in parts of the domain where $f(x)$ is fairly flat, we may find the derivative is near zero even though we are far from the optimum. Instead, we generally monitor $|x_{t+1} - x_t|$ (for the moment, assume x is scalar). We might consider absolute convergence: $|x_{t+1} - x_t| < \epsilon$ or relative convergence, $\frac{|x_{t+1}-x_t|}{|x_t|} < \epsilon$. Relative convergence is appealing because it accounts for the scale of x, but it can run into problems when x_t is near zero, in which case one can use $\frac{|x_{t+1}-x_t|}{|x_t|+\epsilon} < \epsilon$. We would want to account for machine precision in thinking about setting ϵ. For relative convergence a reasonable choice of ϵ would be to use the square root of machine epsilon or about 1×10^{-8}. This is the *reltol* argument in *optim()* in R.

Problems with the optimization may show up in a convergence measure that fails to decrease or cycles (oscillates). Software generally has a stopping rule that stops the algorithm after a fixed number of iterations; these can generally be changed by the user. When an algorithm stops because of the stopping rule before the convergence criterion is met, we say the algorithm has failed to converge. Sometimes we just need to run it longer, but often it indicates a problem with the function being optimized or with your starting value.

For multivariate optimization, we use a distance metric between x_{t+1} and x_t, such as $\|x_{t+1} - x_t\|_p$, often with $p = 1$ or $p = 2$.

4.2 Starting values

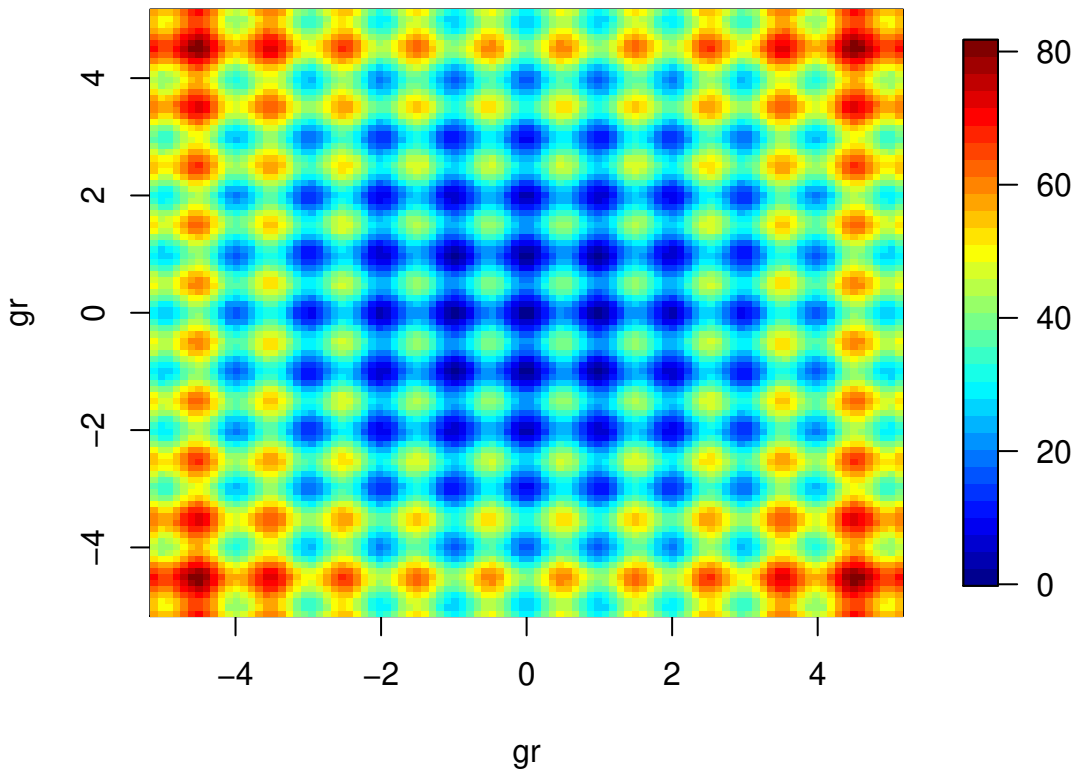
Good starting values are important because they can improve the speed of optimization, prevent divergence or cycling, and prevent finding local optima.

Using random or selected multiple starting values can help with multiple optima (aka multimodality).

Here's a function (the Rastrigin function) with multiple optima that is commonly used for testing methods that claim to work well for multimodal problems. This is a hard function to

optimize with respect to, particularly in higher dimensions (one can do it in higher dimensions than 2 by simply making the x vector longer but having the same structure). In particular Rastrigin with 30 dimensions is considered to be very hard.

```
rastrigin <- function(x) {
        A <- 10
        n <- length(x)
        return(A*n + sum(x^2 - A * cos(2*pi*x)))
}
const <- 5.12
nGrid <- 100
gr <- seq(-const, const, len = nGrid)
xs <- expand.grid(x1 = gr, x2 = gr)
y <- apply(xs, 1, rastrigin)
require(fields)
image.plot(gr, gr, matrix(y, nGrid, nGrid), col=tim.colors(32))
```



One R package that may be useful for multi-modal problems is *DEoptim*, which implements an evolutionary algorithm (genetic algorithms are one kind of evolutionary algorithm). It would be interesting to try an evolutionary algorithm on a test function like this.

4.3 Convergence rates

Let $\epsilon_t = |x_t - x^*|$. If the limit

$$\lim_{t\to\infty} \frac{|\epsilon_{t+1}|}{|\epsilon_t|^\beta} = c$$

exists for $\beta > 0$ and $c \neq 0$, then a method is said to have order of convergence β. This basically measures how big the error at the $t+1$th iteration is relative to that at the tth iteration, with the approximation that $|\epsilon_{t+1}| \approx c|\epsilon_t|^\beta$.

Bisection doesn't formally satisfy the criterion needed to make use of this definition, but roughly speaking it has linear convergence ($\beta = 1$), so the magnitude of the error decreases by a factor of c at each step. Next we'll see that N-R has quadratic convergence ($\beta = 2$), which is fast.

To analyze convergence of N-R, consider a Taylor expansion of the gradient at the minimum, x^*, around the current value, x_t:

$$f'(x^*) = f'(x_t) + (x^* - x_t)f''(x_t) + \frac{1}{2}(x^* - x_t)^2 f'''(\xi_t) = 0,$$

for some $\xi_t \in [x^*, x_t]$. Making use of the N-R update equation: $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$, and some algebra, we have

$$\frac{|x^* - x_{t+1}|}{(x^* - x_t)^2} = \frac{1}{2}\frac{f'''(\xi_t)}{f''(x_t)}.$$

If the limit of the ratio on the right hand side exists and is equal to c:

$$c = \lim_{x_t\to x^*} \frac{1}{2}\frac{f'''(\xi_t)}{f''(x_t)} = \frac{1}{2}\frac{f'''(x^*)}{f''(x^*)}$$

then we see that $\beta = 2$.

If c were one, then we see that if we have k digits of accuracy at t, we'd have $2k$ digits at $t+1$ (e.g., $|\epsilon_t| = 0.01$ results in $|\epsilon_{t+1}| = 0.0001$), which justifies the characterization of quadratic convergence being fast. In practice c will moderate the rate of convergence. The smaller c the better, so we'd like to have the second derivative be large and the third derivative be small. The expression also indicates we'll have a problem if $f''(x_t) = 0$ at any point [think about what this corresponds to graphically - what is our next step when $f''(x_t) = 0$?]. The characteristics of the derivatives determine the domain of attraction (the region in which we'll converge rather than

diverge) of the minimum.

Givens and Hoeting show that using the secant-based approximation to the second derivative in N-R has order of convergence, $\beta \approx 1.62$.

Here's an example of convergence comparing bisection and N-R:

```
options(digits = 10)
f <- function(x) cos(x)
fp <- function(x) -sin(x)
fpp <- function(x) -cos(x)
xstar <- pi # known minimum

## N-R
x0 <- 2
xvals <- c(x0,rep(NA,9))
for(t in 2:10){
        xvals[t] <- xvals[t-1] - fp(xvals[t-1]) / fpp(xvals[t-1])
}
print(xvals)

##  [1] 2.000000000 4.185039863 2.467893675 3.266186278 3.140943912
##  [6] 3.141592654 3.141592654 3.141592654 3.141592654 3.141592654

## bisection
bisecStep <- function(interval, fp){
        xt <- mean(interval)
        if(fp(interval[1]) * fp(xt) <= 0) interval[2] <- xt else interval[1]
        return(interval)
}
nIt <- 30
a0 <- 2; b0 <- (3*pi/2) - (xstar - a0)
## have b0 be as far from min as a0 for fair comparison with N-R
interval <- matrix(NA, nr = nIt, nc = 2)
interval[1, ] <- c(a0, b0)
for(t in 2:nIt){
        interval[t, ] <- bisecStep(interval[t-1, ], fp)
}
rowMeans(interval)
```

```
##  [1] 2.785398163 3.178097245 2.981747704 3.079922475 3.129009860
##  [6] 3.153553552 3.141281706 3.147417629 3.144349668 3.142815687
## [11] 3.142048697 3.141665201 3.141473454 3.141569328 3.141617264
## [16] 3.141593296 3.141581312 3.141587304 3.141590300 3.141591798
## [21] 3.141592547 3.141592922 3.141592734 3.141592641 3.141592687
## [26] 3.141592664 3.141592652 3.141592658 3.141592655 3.141592654
```

5 Multivariate optimization

Optimizing as the dimension of the space gets larger becomes increasingly difficult. First we'll discuss the idea of profiling to reduce dimensionality and then we'll talk about various numerical techniques, many of which build off of Newton's method.

5.1 Profiling

A core technique for likelihood optimization is to analytically maximize over any parameters for which this is possible. Suppose we have two sets of parameters, θ_1 and θ_2, and we can analytically maximize w.r.t θ_2. This will give us $\hat{\theta}_2(\theta_1)$, a function of the remaining parameters over which analytic maximization is not possible. Plugging in $\hat{\theta}_2(\theta_1)$ into the objective function (in this case generally the likelihood or log likelihood) gives us the profile (log) likelihood solely in terms of the obstinant parameters. For example, suppose we have the regression likelihood with correlated errors:

$$Y \sim \mathcal{N}(X\beta, \sigma^2\Sigma(\rho)),$$

where $\Sigma(\rho)$ is a correlation matrix that is a function of a parameter, ρ. The maximum w.r.t. β is easily seen to be the GLS estimator $\hat{\beta}(\rho) = (X^\top\Sigma(\rho)^{-1}X)^{-1}X^\top\Sigma(\rho)^{-1}Y$. In general such a maximum is a function of all of the other parameters, but conveniently it's only a function of ρ here. This gives us the initial profile likelihood

$$\frac{1}{(\sigma^2)^{n/2}|\Sigma(\rho)|^{1/2}} \exp\left(-\frac{(Y - X\hat{\beta}(\rho))^{-\top}\Sigma(\rho)^{-1}(Y - X\hat{\beta}(\rho))}{2\sigma^2}\right).$$

We then notice that the likelihood is maximized w.r.t. σ^2 at

$$\hat{\sigma^2}(\rho) = \frac{(Y - X\hat{\beta}(\rho))^\top\Sigma(\rho)^{-1}(Y - X\hat{\beta}(\rho))}{n}.$$

This gives us the final profile likelihood,

$$\frac{1}{|\Sigma(\rho)|^{1/2}} \frac{1}{(\hat{\sigma^2}(\rho))^{n/2}} \exp(-\frac{1}{2}n),$$

a function of ρ only, for which numerical optimization is much simpler.

5.2 Newton-Raphson (Newton's method)

For multivariate x we have the Newton-Raphson update $x_{t+1} = x_t - f''(x_t)^{-1} f'(x_t)$, or in our other notation,

$$x_{t+1} = x_t - H_f(x_t)^{-1} \nabla f(x_t).$$

In class we'll use the demo code in the Unit 11 code file (not shown here) for an example of finding the nonlinear least squares fit to some weight loss data to fit the model (but note that technically speaking one can use profiling in this case, so it's not a perfect example):

$$Y_i = \beta_0 + \beta_1 2^{-t_i/\beta_2} + \epsilon_i.$$

Some of the things we need to worry about with Newton's method in general about are (1) good starting values, (2) positive definiteness of the Hessian, and (3) avoiding errors in deriving the derivatives.

A note on the positive definiteness: since the Hessian may not be positive definite (although it may well be, provided the function is approximately locally quadratic), one can consider modifying the Cholesky decomposition of the Hessian to enforce positive definiteness by adding diagonal elements to H_f as necessary.

5.3 Fisher scoring variant on N-R

The Fisher information (FI) is the expected value of the outer product of the gradient of the log-likelihood with itself

$$I(\theta) = E_f(\nabla f(y) \nabla f(y)^\top),$$

where the expected value is with respect to the data distribution. Under regularity conditions (true for exponential families), the expectation of the Hessian of the log-likelihood is minus the Fisher information, $E_f H_f(y) = -I(\theta)$. We get the observed Fisher information by plugging the data values into either expression instead of taking the expected value.

Thus, standard N-R can be thought of as using the observed Fisher information to find the updates. Instead, if we can compute the expectation, we can use minus the FI in place of the

Hessian. The result is the Fisher scoring (FS) algorithm. Basically instead of using the Hessian for a given set of data, we are using the FI, which we can think of as the average Hessian over repeated samples of data from the data distribution. FS and N-R have the same convergence properties (i.e., quadratic convergence) but in a given problem, one may be computationally or analytically easier. Givens and Hoeting comment that FS works better for rapid improvements at the beginning of iterations and N-R better for refinement at the end.

$$\begin{aligned}(NR):\ \theta_{t+1} &= \theta_t - H_f(\theta_t)^{-1}\nabla f(\theta_t) \\ (FS):\ \theta_{t+1} &= \theta_t + I(\theta_t)^{-1}\nabla f(\theta_t)\end{aligned}$$

In class we'll use the demo code in the Unit 11 code file (not shown here) to try out Fisher scoring in the weight loss example.

The Gauss-Newton algorithm for nonlinear least squares involves using the FI in place of the Hessian in determining a Newton-like step. *nls()* in R uses this approach. Note that this is not exactly the same updating as our manual coding of FS for the weight loss example.

Connections between statistical uncertainty and ill-conditionedness When either the observed or expected FI matrix is nearly singular this means we have a small eigenvalue in the inverse covariance (the precision), which means a large eigenvalue in the covariance matrix. This indicates some linear combination of the parameters has low precision (high variance), and that in that direction the likelihood is nearly flat. As we've seen with N-R, convergence slows with shallow gradients, and we may have numerical problems in determining good optimization steps when the likelihood is sufficiently flat. So convergence problems and statistical uncertainty go hand in hand. One, but not the only, example of this occurs when we have nearly collinear regressors.

5.4 IRLS (IWLS) for GLMs

As most of you know, iterative reweighted least squares (also called iterative weighted least squares) is the standard method for estimation with GLMs. It involves linearizing the model and using working weights and working variances and solving a weighted least squares (WLS) problem (the generic WLS solution is $\hat{\beta} = (X^\top W X)^{-1} X^\top W Y$).

Exponential families can be expressed as

$$f(y;\theta,\phi) = \exp((y\theta - b(\theta))/a(\phi) + c(y,\phi)),$$

with $E(Y) = b'(\theta)$ and $\text{Var}(Y) = b''(\theta)$. If we have a GLM in the canonical parameterization (log link for Poisson data, logit for binomial), we have the natural parameter θ equal to the linear

predictor, $\theta = \eta$. A standard linear predictor would simply be $\eta = X\beta$.

Considering N-R for a GLM in the canonical parameterization (and ignoring $a(\phi)$, which is one for logistic and Poisson regression), we find that the gradient is the inner product of the covariates and a residual vector, $\nabla f(\beta) = (Y - E(Y))^\top X$, and the Hessian is $\nabla^2 f(\beta) = -X^\top W X$ where W is a diagonal matrix with $\{\text{Var}(Y_i)\}$ on the diagonal (the working weights). Note that both $E(Y)$ and the variances in W depend on β, so these will change as we iteratively update β. Therefore, the N-R update is

$$\beta_{t+1} = \beta_t + (X^\top W_t X)^{-1} X^\top (Y - E(Y)_t)$$

where $E(Y)_t$ and W_t are the values at the current parameter estimate, β_t . For example, for logistic regression (here with $n_i = 1$), $W_{t,ii} = p_{ti}(1 - p_{ti})$ and $E(Y)_{ti} = p_{ti}$ where $p_{ti} = \frac{\exp(X_i^\top \beta_t)}{1+\exp(X_i^\top \beta_t)}$. In the canonical parameterization of a GLM, the Hessian does not depend on the data, so the observed and expected FI are the same, and therefore N-R and FS are the same.

The update above can be rewritten in the standard form of IRLS as a WLS problem,

$$\beta_{t+1} = (X^\top W_t X)^{-1} X^\top W_t \tilde{Y}_t,$$

where the so-called working observations are $\tilde{Y}_t = X\beta_t + W_t^{-1}(Y - E(Y)_t)$. Note that these are on the scale of the linear predictor.

While Fisher scoring is standard for GLMs, you can also use general purpose optimization routines.

IRLS is a special case of the general Gauss-Newton method for nonlinear least squares.

5.5 Descent methods and Newton-like methods

More generally a Newton-like method has updates of the form

$$x_{t+1} = x_t - \alpha_t M_t^{-1} f'(x_t).$$

We can choose M_t in various ways, including as an approximation to the second derivative.

This opens up several possibilities:

1. using more computationally efficient approximations to the second derivative,
2. avoiding steps that do not go in the correct direction (i.e., go uphill when minimizing), and
3. scaling by α_t so as not to step too far.

Let's consider a variety of strategies.

5.5.1 Descent methods

The basic strategy is to choose a good direction and then choose the longest step for which the function continues to decrease. Suppose we have a direction, p_t. Then we need to move $x_{t+1} = x_t + \alpha_t p_t$, where α_t is a scalar, choosing a good α_t. We might use a line search (e.g., bisection or golden section search) to find the local minimum of $f(x_t + \alpha_t p_t)$ with respect to α_t. However, we often would not want to run to convergence, since we'll be taking additional steps anyway.

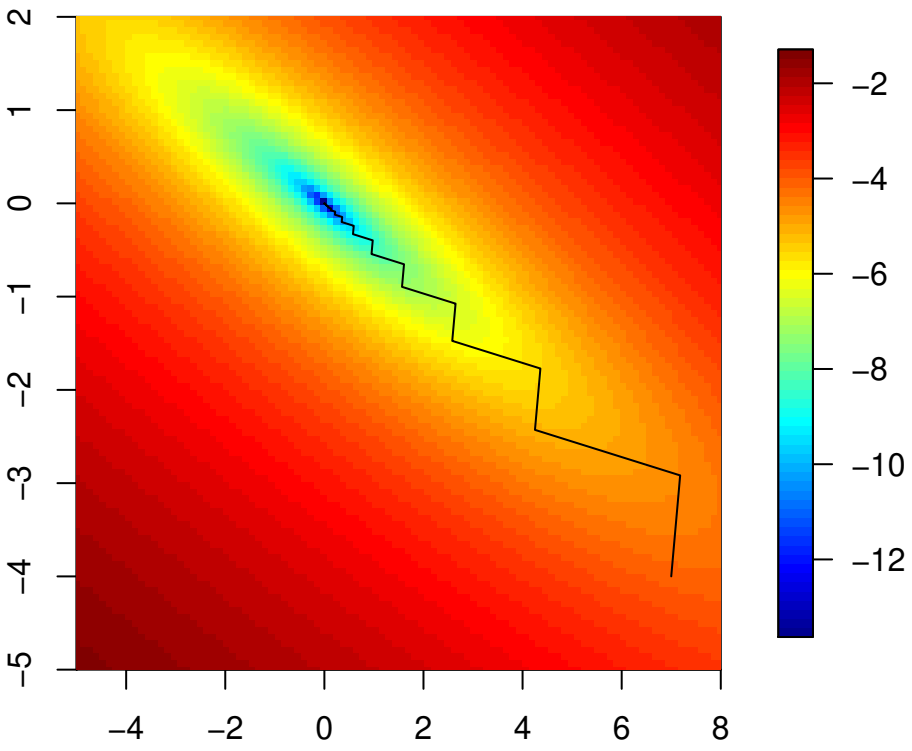
Steepest descent chooses the direction as the steepest direction downhill, setting $M_t = I$, since the gradient gives the steepest direction uphill (the negative sign in the equation below has us move directly downhill rather than directly uphill). Given the direction, we want to scale the step

$$x_{t+1} = x_t - \alpha_t f'(x_t)$$

where the contraction, or step length, parameter α_t is chosen sufficiently small to ensure that we descend, via some sort of line search. The critical downside to steepest descent is that when the contours are elliptical, it tends to zigzag; here's an example. Note that I do a full line search (using the golden section method via *optimize()*) at each step in the direction of steepest descent - this is generally computationally wasteful, but I just want to illustrate how steepest descent can go wrong, even if you go the "right" amount in each direction.

```
par(mai = c(.5,.4,.1,.4))
f <- function(x){
        x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000
}
fp <- function(x){
        c(2 * x[1]/1000 + 4 * x[2]/1000,
        4 * x[1]/1000 + 10 * x[2]/1000)
}
lineSearch <- function(alpha, xCurrent, direction, FUN){
        newx <- xCurrent + alpha * direction
        FUN(newx)
}
nIt <- 50
xvals <- matrix(NA, nr = nIt, nc = 2)
xvals[1, ] <- c(7, -4)
for(t in 2:50){
        newalpha <- optimize(lineSearch, interval = c(-5000, 5000),
```

```
                xCurrent = xvals[t-1, ], direction = fp(xvals[t-1, ]),
                FUN = f)$minimum
        xvals[t, ] <- xvals[t-1, ] + newalpha * fp(xvals[t-1, ])
}
x1s <- seq(-5, 8, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
## plot f(x) surface on log scale
image.plot(x1s, x2s, matrix(log(fx), 100, 100),
        xlim = c(-5, 8), ylim = c(-5,2))
lines(xvals) ## overlay optimization path
```




```
## kind of slow
```

If the contours are circular, steepest descent works well. Newton's method deforms elliptical contours based on the Hessian. Another way to think about this is that steepest descent does not take account of the rate of change in the gradient, while Newton's method does.

The general descent algorithm is

$$x_{t+1} = x_t - \alpha_t M_t^{-1} f'(x_t),$$

where M_t is generally chose to approximate the Hessian and α_t allows us to adjust the step in a smart way. Basically, since the negative gradient tells us the direction that descends (at least within a small neighborhood), if we don't go too far, we should be fine and should work our way downhill. One can work this out formally using a Taylor approximation to $f(x_{t+1}) - f(x_t)$ and see that we make use of M_t being positive definite. (Unfortunately backtracking with positive definite M_t does not give a theoretical guarantee that the method will converge. We also need to make sure that the steps descend sufficiently quickly and that the algorithm does not step along a level contour of f.)

The conjugate gradient algorithm for iteratively solving large systems of equations is all about choosing the direction and the step size in a smart way given the optimization problem at hand.

5.5.2 Quasi-Newton methods such as BFGS

Other replacements for the Hessian matrix include estimates that do not vary with t and finite difference approximations. When calculating the Hessian is expensive, it can be very helpful to substitute an approximation.

A basic finite difference approximation requires us to compute finite differences in each dimension, but this could be computationally burdensome. A more efficient strategy for choosing M_{t+1} is to (1) make use of M_t and (2) make use of the most recent step to learn about the curvature of $f'(x)$ in the direction of travel. One approach is to use a rank one update to M_t.

A basic strategy is to choose M_{t+1} such that the secant condition is satisfied:

$$M_{t+1}(x_{t+1} - x_t) = \nabla f(x_{t+1}) - \nabla f(x_t),$$

which is motivated by the fact that the secant approximates the gradient in the direction of travel. Basically this says to modify M_t in such a way that we incorporate what we've learned about the gradient from the most recent step. M_{t+1} is not fully determined based on this, and we generally impose other conditions, in particular that M_{t+1} is symmetric and positive definite. Defining $s_t = x_{t+1} - x_t$ and $y_t = \nabla f(x_{t+1}) - \nabla f(x_t)$, the unique, symmetric rank one update (why is the following a rank one update?) that satisfies the secant condition is

$$M_{t+1} = M_t + \frac{(y_t - M_t s_t)(y_t - M_t s_t)^\top}{(y_t - M_t s_t)^\top s_t}.$$

If the denominator is positive, M_{t+1} may not be positive definite, but this is guaranteed for non-positive values of the denominator. One can also show that one can achieve positive definiteness by shrinking the denominator toward zero sufficiently.

A standard approach to updating M_t is a commonly-used rank two update that generally results

in M_{t+1} being positive definite is

$$M_{t+1} = M_t - \frac{M_t s_t (M_t s_t)^\top}{s_t^\top M_t s_t} + \frac{y_t y_t^\top}{s_t^\top y_t},$$

which is known as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update. This is one of the methods used in R in *optim()*.

Question: how can we update M_t^{-1} to M_{t+1}^{-1} efficiently? It turns out there is a way to update the Cholesky of M_t efficiently and this is a better approach than updating the inverse.

The order of convergence of quasi-Newton methods is generally slower than the quadratic convergence of N-R because of the approximations but still faster than linear. In general, quasi-Newton methods will do much better if the scales of the elements of x are similar. Lange suggests using a starting point for which one can compute the expected information, to provide a good starting value M_0.

Note that for estimating a covariance based on the numerical information matrix, we would not want to rely on M_t from the final iteration, as the approximation may be poor. Rather we would spend the effort to better estimate the Hessian directly at x^*.

5.5.3 Stochastic gradient descent

Stochastic gradient descent (SGD) is the hot method in machine learning, commonly used for fitting deep neural networks. It allows you to optimize an objective function with respect to what is often a very large number of parameters even when the data size is huge.

Gradient descent is a simplification of Newton's method that does not rely on the second derivative, but rather chooses the direction using the gradient and then a step size, α_t:

$$x_{t+1} = x_t - \alpha_t f'(x_t)$$

The basic idea of stochastic gradient descent is to replace the gradient with a function whose expected value is the gradient, $E(g(x_t)) = f'(x_t)$:

$$x_{t+1} = x_t - \alpha_t g(x_t)$$

Thus on average we should go in a good (downhill) direction. Given that we know that strictly following the gradient can lead to slow convergence, it makes some intuitive sense that we could still do ok without using the exact gradient. One can show formally that SGD will converge for convex functions.

SGD can be used in various contexts, but the common one we will focus on is when

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} f_i(x) \\ f'(x) &= \sum_{i=1}^{n} f_i'(x) \end{aligned}$$

for large n. Thus calculation of the gradient is $O(n)$, and we may not want to incur that computational cost. How could we implement SGD in such a case? At each iteration we could randomly choose an observation and compute the contribution to the gradient from that data point, or we could choose a random subset of the data (this is *mini-batch SGD*), or there are variations where we systematically cycle through the observations or cycle through subsets. However, in some situations, convergence is actually much faster when using randomness. And if the data are ordered in some meaningful way we definitely do not want to cycle through the observations in that order, as this can result in a biased estimate of the gradient and slow convergence. So one generally randomly shuffles the data before starting SGD. Note that using subsets rather than individual observations is likely to be more effective as it can allow us to use optimized matrix/vector computations.

How should one choose the step size, α_t (also called the learning rate)? One might think that as one gets close to the optimum, if one isn't careful, one might simply bounce around near the optimum in a random way, without actually converging to the optimum. So intuition suggests that α_t should decrease with t. Some choices of step size have included:

- $\alpha_t = 1/t$
- set a schedule, such that for T iterations, $\alpha_t = \alpha$, then for the next T, $\alpha_t = \alpha\gamma$, then for the next T, $\alpha_t = \alpha\gamma^2$. A heuristic is for $\gamma \in (0.8, 0.9)$.
- run with $\alpha_t = \alpha$ for T iterations, then with $\alpha_t = \alpha/2$ for $2T$, then with $\alpha_t = \alpha/4$ for $4T$ and so forth.

5.6 Coordinate descent (Gauss-Seidel)

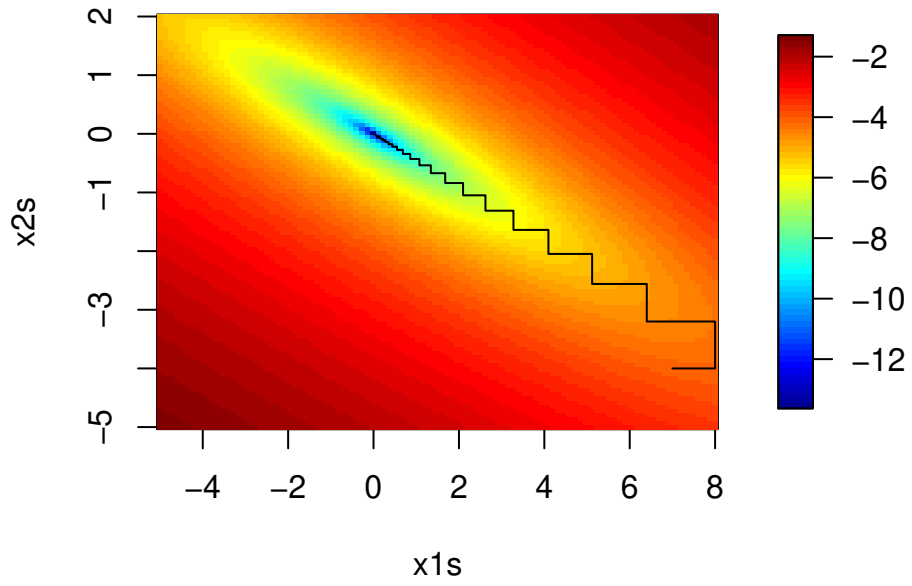
Gauss-Seidel is also known a back-fitting or cyclic coordinate descent. The basic idea is to work element by element rather than having to choose a direction for each step. For example backfitting used to be used to fit generalized additive models of the form $E(Y) = f_1(z_1)+f_2(z_2)+\ldots+f_p(z_p)$.

The basic strategy is to consider the jth component of $f'(x)$ as a univariate function of x_j only and find the root, $x_{j,t+1}$ that gives $f_j'(x_{j,t+1}) = 0$. One cycles through each element of x to

complete a single cycle and then iterates. The appeal is that univariate root-finding/minimization is easy, often more stable than multivariate, and quick.

However, Gauss-Seidel can zigzag, since you only take steps in one dimension at a time, as we see here.

```
f <- function(x){
        return(x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000)
}
f1 <- function(x1, x2){ # f(x) as a function of x1
        return(x1^2/1000 + 4*x1*x2/1000 + 5*x2^2/1000)
}
f2 <- function(x2, x1){ # f(x) as a function of x2
        return(x1^2/1000 + 4*x1*x2/1000 + 5*x2^2/1000)
}
x1s <- seq(-5, 8, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
image.plot(x1s, x2s, matrix(log(fx), 100, 100))
nIt <- 49
xvals <- matrix(NA, nr = nIt, nc = 2)
xvals[1, ] <- c(7, -4)
## 5, -10
for(t in seq(2, nIt, by = 2)){
        newx1 <- optimize(f1, x2 = xvals[t-1, 2], interval = c(-40, 40))$mi
        xvals[t, ] <- c(newx1, xvals[t-1, 2])
        newx2 <- optimize(f2, x1 = newx1, interval = c(-40, 40))$minimum
        xvals[t+1, ] <- c(newx1, newx2)
}
lines(xvals)
```



In the notes for Unit 9 on linear algebra, I discussed the use of Gauss-Seidel to iteratively solve $Ax = b$ in situations where factorizing A (which of course is $O(n^3)$) is too computationally expensive.

The lasso The *lasso* uses an L1 penalty in regression and related contexts. A standard formulation for the lasso in regression is to minimize

$$\|Y - X\beta\|_2^2 + \lambda \sum_j |\beta_j|$$

to find $\hat{\beta}(\lambda)$ for a given value of the penalty parameter, λ. A standard strategy to solve this problem is to use coordinate descent, either cyclically, or by using directional derivatives to choose the coordinate likely to decrease the objective function the most (a greedy strategy). We need to use directional derivatives because the penalty function is not differentiable, but does have directional derivatives in each direction. The directional derivative of the objective function for β_j is

$$-2 \sum_i x_{ij}(Y_i - X_i^\top \beta) \pm \lambda$$

where we add λ if $\beta_j \geq 0$ and you subtract λ if $\beta_j < 0$. If $\beta_{j,t}$ is 0, then a step in either direction contributes $+\lambda$ to the derivative as the contribution of the penalty.

Once we have chosen a coordinate, we set the directional derivative to zero and solve for β_j to obtain $\beta_{j,t+1}$.

The *glmnet* package in R (described in this Journal of Statistical Software paper) implements such optimization for a variety of penalties in linear model and GLM settings, including the lasso. This Mittal et al. paper describes similar optimization for survival analysis with very large p, exploiting sparsity in the X matrix for computational efficiency; note that they do not use Newton-Raphson because the matrix operations are infeasible computationally.

One nice idea that is used in lasso and related settings is the idea of finding the regression coefficients for a variety of values of λ, combined with "warm starts". A general approach is to start with a large value of λ for which all the coefficients are zero and then decrease λ. At each new value of λ, use the estimated coefficients from the previous value as the starting values. This should allow for fast convergence and gives what is called the "solution path". Often λ is chosen based on cross-validation.

The LARS (least angle regression) algorithm uses a similar strategy that allows one to compute $\hat{\beta}_\lambda$ for all values of λ at once.

The lasso can also be formulated as the constrained minimization of $\|Y - X\beta\|_2^2$ s.t. $\sum_j |\beta_j| \leq c$, with c now playing the role of the penalty parameter. Solving this minimization problem would take us in the direction of quadratic programming, a special case of convex programming, discussed in Section 9.

5.7 Nelder-Mead

This approach avoids using derivatives or approximations to derivatives. This makes it robust, but also slower than Newton-like methods. The basic strategy is to use a simplex, a polytope of $p+1$ points in p dimensions (e.g., a triangle when searching in two dimensions, tetrahedron in three dimensions...) to explore the space, choosing to shift, expand, or contract the polytope based on the evaluation of f at the points.

The algorithm relies on four tuning factors: a reflection factor, $\alpha > 0$; an expansion factor, $\gamma > 1$; a contraction factor, $0 < \beta < 1$; and a shrinkage factor, $0 < \delta < 1$. First one chooses an initial simplex: $p+1$ points that serve as the vertices of a convex hull.

1. Evaluate and order the points, $x_1, \ldots, x_{p+1}$ based on $f(x_1) \leq \ldots \leq f(x_{p+1})$. Let $\bar{x}$ be the average of the first p x's.

2. (Reflection) Reflect x_{p+1} across the hyperplane (a line when $p+1=3$) formed by the other points to get x_r, based on α.

    - $x_r = (1+\alpha)\bar{x} - \alpha x_{p+1}$

3. If $f(x_r)$ is between the best and worst of the other points, the iteration is done, with x_r replacing x_{p+1}. We've found a good direction to move.

4. (Expansion) If $f(x_r)$ is better than all of the other points, expand by extending x_r to x_e based on γ, because this indicates the optimum may be further in the direction of reflection. If $f(x_e)$ is better than $f(x_r)$, use x_e in place of x_{p+1}. If not, use x_r. The iteration is done.

    - $x_e = \gamma x_r + (1 - \gamma)\bar{x}$

5. If $f(x_r)$ is worse than all the other points, but better than $f(x_{p+1})$, let $x_h = x_r$. Otherwise $f(x_r)$ is worse than $f(x_{p+1})$ so let $x_h = x_{p+1}$. In either case, we want to concentrate our polytope toward the other points.

    (a) (Contraction) Contract x_h toward the hyperplane formed by the other points, based on β, to get x_c. If the result improves upon $f(x_h)$ replace x_{p+1} with x_c. Basically, we haven't found a new point that is better than the other points, so we want to contract the simplex away from the bad point.

    - $x_c = \beta x_h + (1 - \beta)\bar{x}$

    (b) (Shrinkage) Otherwise (if x_c is not better than x_{p+1}) shrink the simplex toward x_1. Basically this suggests our step sizes are too large and we should shrink the simplex, shrinking towards the best point.

    - $x_i = \delta x_i + (1 - \delta)x_1$for $i = 2, \ldots, p + 1$

Convergence is assessed based on the sample variance of the function values at the points, the total of the norms of the differences between the points in the new and old simplexes, or the size of the simplex. In class we'll work through some demo code (not shown here) that illustrates the individual steps in an iteration of Nelder-Mead.

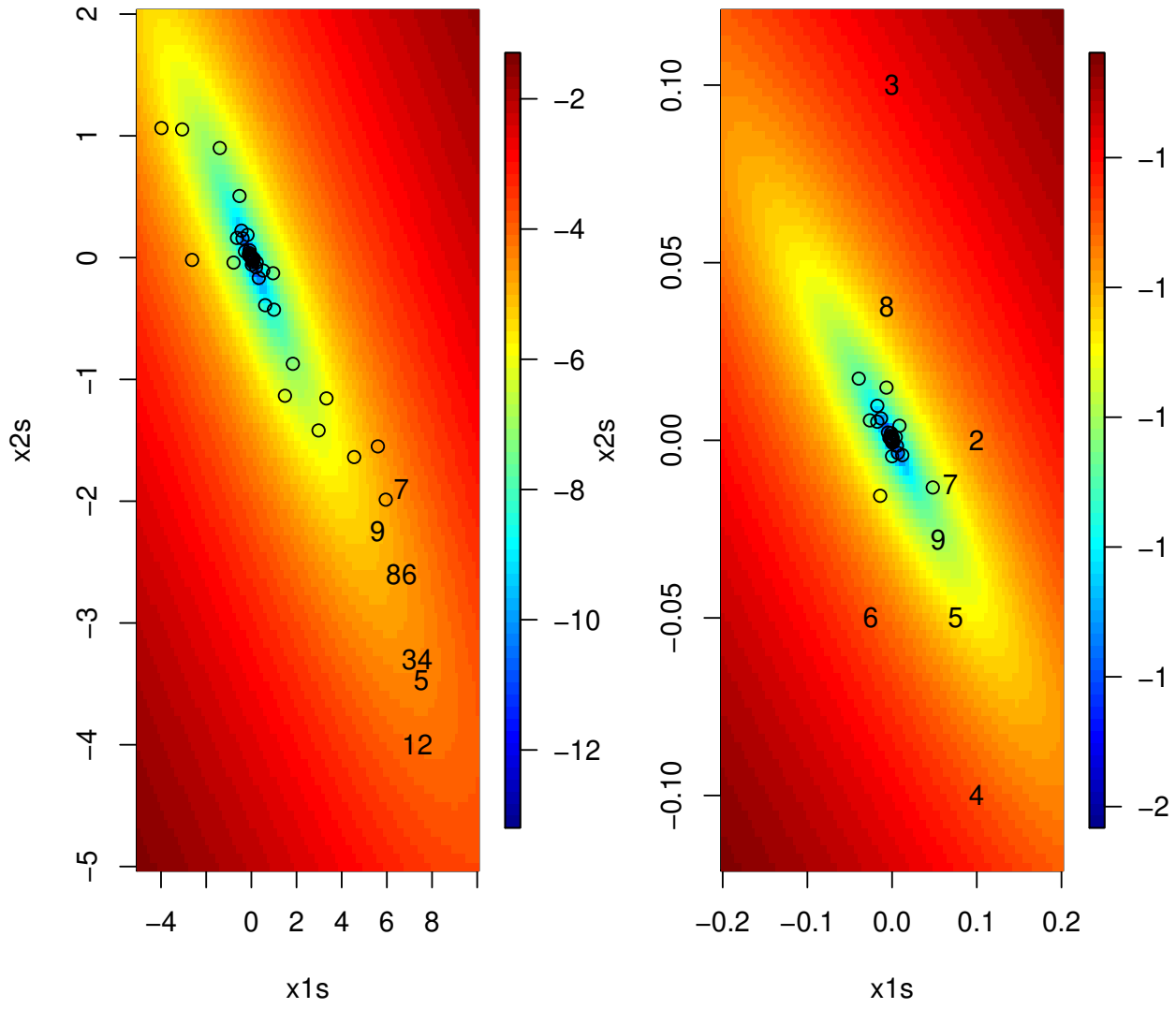
We can see the points at which the function was evaluated in the same quadratic example we saw in previous sections. The left hand panel shows the steps from a starting point somewhat far from the optimum, with the first 9 points numbered. In this case, we start with points 1, 2, and 3. Point 4 is a reflection. At this point, it looks like point 5 is a contraction but that doesn't exactly follow the algorithm above, so perhaps the algorithm as implemented is a bit different than as described above. The new set is (2, 3, 4). Then point 6 and point 7 are reflection and expansion steps and the new set is (3, 4, 6). Points 8 and 9 are again reflection and expansion steps. The right hand panel shows the steps from a starting point near (actually at) the optimum. Points 4 and 5 are reflection and expansion steps, with the next set being (1, 2, 5). Now step 6 is a reflection but it is the worst of all the points, so point 7 is a contraction of point 2 giving the next set (1, 5, 7). Point 8 is then a reflection and point 9 is a contraction of point 5.

```r
f <- function(x, plot = TRUE){
    if(plot && cnt < 10) {
        points(x[1], x[2], pch = as.character(cnt))
        if(cnt < 10) cnt <<- cnt + 1 else cnt <<- 1
    } else if(plot) points(x[1], x[2])
    # if(plot) print(c(x,x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000))
    return(x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000)
}

library(fields)
par(mfrow = c(1,2))

x1s <- seq(-5, 10, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f, FALSE)
cnt <- 1
image.plot(x1s, x2s, matrix(log(fx), 100, 100))
init <- c(7, -4)
optim(init, f, method = "Nelder-Mead")
```

```
## $par
## [1] -0.0006112564567  0.0001513538132
##
## $value
## [1] 1.181103575e-10
##
## $counts
## function gradient
##       65       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
x1s <- seq(-.2, .2, len = 100); x2s = seq(-.12, .12, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f, FALSE)
```

```
cnt <- 1
image.plot(x1s, x2s, matrix(log(fx), 100, 100))
init <- c(-0, 0)
optim(init, f, method = "Nelder-Mead")
```




```
## $par
## [1] 0 0
##
## $value
```

```
## [1] 0
##
## $counts
## function gradient
##       77       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Here's an online graphical illustration of Nelder-Mead.

This is the default in *optim()* in R, however it is relatively slow, so you may want to try one of the alternatives, such as BFGS.

5.8 Simulated annealing (SA)

Simulated annealing is a *stochastic* descent algorithm, unlike the deterministic algorithms we've already discussed. It has a couple critical features that set it aside from other approaches. First, uphill moves are allowed; second, whether a move is accepted is stochastic, and finally, as the iterations proceed the algorithm becomes less likely to accept uphill moves.

Assume we are minimizing a negative log likelihood as a function of θ, $f(\theta)$.

The basic idea of simulated annealing is that one modifies the objective function, f in this case, to make it less peaked at the beginning, using a "temperature" variable that changes over time. This helps to allow moves away from local minima, when combined with the ability to move uphill. The name comes from an analogy to heating up a solid to its melting temperature and cooling it slowly - as it cools the atoms go through rearrangements and slowly freeze into the crystal configuration that is at the lowest energy level.

Here's the algorithm. We divide up iterations into stages, $j = 1, 2, \ldots$ in which the temperature variable, τ_j, is constant. Like MCMC, we require a proposal distribution to propose new values of θ.

1. Propose to move from θ_t to $\tilde{\theta}$ from a proposal density, $g_t(\cdot|\theta_t)$, such as a normal distribution centered at θ_t.

2. Accept $\tilde{\theta}$ as θ_{t+1} according to the probability $\min(1, \exp((f(\theta_t) - f(\tilde{\theta}))/\tau_j)$ - i.e., accept if a uniform random deviate is less than that probability. Otherwise set $\theta_{t+1} = \theta_t$. Notice that

for larger values of τ_j the differences between the function values at the two locations are reduced (just like a large standard deviation spreads out a distribution). So the exponentiation smooths out the objective function when τ_j is large.

3. Repeat steps 1 and 2 m_j times.

4. Increment the temperature and cooling schedule: $\tau_j = \alpha(\tau_{j-1})$ and $m_j = \beta(m_{j-1})$. Back to step 1.

The temperature should slowly decrease to 0 while the number of iterations, m_j, should be large. Choosing these 'schedules' is at the core of implementing SA. Note that we always accept downhill moves in step 2 but we sometimes accept uphill moves as well.

For each temperature, SA produces an MCMC based on the Metropolis algorithm. So if m_j is long enough, we should sample from the stationary distribution of the Markov chain, $\exp(-f(\theta)/\tau_j))$. Provided we can move between local minima, the chain should gravitate toward the global minima because these are increasingly deep (low values) relative to the local minima as the temperature drops. Then as the temperature cools, θ_t should get trapped in an increasingly deep well centered on the global minimum. There is a danger that we will get trapped in a local minimum and not be able to get out as the temperature drops, so the temperature schedule is quite important in trying to avoid this.

A wide variety of schedules have been tried. One approach is to set $m_j = 1 \forall j$ and $\alpha(\tau_{j-1}) = \frac{\tau_{j-1}}{1+a\tau_{j-1}}$ for a small a. For a given problem it can take a lot of experimentation to choose τ_0 and m_0 and the values for the scheduling functions. For the initial temperature, it's a good idea to choose it large enough that $\exp((f(\theta_i) - f(\theta_j))/\tau_0) \approx 1$ for any pair $\{\theta_i, \theta_j\}$ in the domain, so that the algorithm can visit the entire space initially.

Simulated annealing can converge slowly. Multiple random starting points or stratified starting points can be helpful for finding a global minimum. However, given the slow convergence, these can also be computationally burdensome.

6 Basic optimization in R

6.1 Core optimization functions

R has several optimization functions.

- *optimize()* is good for 1-d optimization: "The method used is a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions."

- Another option is *uniroot()* for finding the zero of a function, which you can use to minimize a function if you can compute the derivative.
- For more than one variable, *optim()* uses a variety of optimization methods including the robust Nelder-Mead method, the BFGS quasi-Newton method and simulated annealing. You can choose which method you prefer and can try multiple methods. You can supply a gradient function to *optim()* for use with the Newton-related methods but it can also calculate numerical derivatives on the fly. You can have *optim()* return the Hessian at the optimum (based on a numerical estimate), which then allows straighforward calculation of asymptotic variances based on the information matrix.
- Also for multivariate optimization, *nlm()* uses a Newton-style method, for which you can supply analytic gradient and Hessian, or it will estimate these numerically. *nlm()* can also return the Hessian at the optimum.
- The *optimx* package provides *optimx()*, which is a wrapper for a variety of optimization methods (including many of those in *optim()*, as well as *nlm()*. One nice feature is that it allow you to use multiple methods in the same function call.

In the demo code (not shown here), we'll work our way through a real example of optimizing a likelihood for some climate data on extreme precipitation.

6.2 Various considerations in using the R functions

As we've seen, initial values are important both for avoiding divergence (e.g., in N-R), for increasing speed of convergence, and for helping to avoid local optima. So it is well worth the time to try to figure out a good starting value or multiple starting values for a given problem.

Scaling can be important. One useful step is to make sure the problem is well-scaled, namely that a unit step in any parameter has a comparable change in the objective function, preferably approximately a unit change at the optimum. *optim()* allows you to supply scaling information through the *parscale* component of the *control* argument. Basically if x_j is varying at p orders of magnitude smaller than the other xs, we want to reparameterize to $x_j^* = x_j \cdot 10^p$ and then convert back to the original scale after finding the answer. Or we may want to work on the log scale for some variables, reparameterizing as $x_j^* = \log(x_j)$. We could make such changes manually in our expression for the objective function or make use of arguments such as *parscale*.

If the function itself gives very large or small values near the solution, you may want to rescale the entire function to avoid calculations with very large or small numbers. This can avoid problems such as having apparent convergence because a gradient is near zero, simply because the scale of the function is small. In *optim()* this can be controlled with the *fnscale* component of *control*.

Always consider your answer and make sure it makes sense, in particular that you haven't 'converged' to an extreme value on the boundary of the space.

Venables and Ripley suggest that it is often worth supplying analytic first derivatives rather than having a routine calculate numerical derivatives but not worth supplying analytic second derivatives. As we'll see in Unit 12, R can do symbolic (i.e., analytic) differentiation to find first and second derivatives using *deriv()*.

In general for software development it's obviously worth putting more time into figuring out the best optimization approach and supplying derivatives. For a one-off analysis, you can try a few different approaches and assess sensitivity.

The nice thing about likelihood optimization is that the asymptotic theory tells us that with large samples, the likelihood is approximately quadratic (i.e., the asymptotic normality of MLEs), which makes for a nice surface over which to do optimization. When optimizing with respect to variance components and other parameters that are non-negative, one approach to dealing with the constraints is to optimize with respect to the log of the parameter.

7 Combinatorial optimization over discrete spaces

Many statistical optimization problems involve continuous domains, but sometimes there are problems in which the domain is discrete. Variable selection is an example of this.

Simulated annealing can be used for optimizing in a discrete space. Another approach uses *genetic algorithms*, in which one sets up the dimensions as loci grouped on a chromosome and has mutation and crossover steps in which two potential solutions reproduce. An example would be in high-dimensional variable selection.

Stochastic search variable selection is a popular Bayesian technique for variable selection that involves MCMC.

8 Convexity

Many optimization problems involve (or can be transformed into) convex functions. Convex optimization (also called convex programming) is a big topic and one that we'll only brush the surface of in Sections 8 and 9. The goal here is to give you enough of a sense of the topic that you know when you're working on a problem that might involve convex optimization, in which case you'll need to go learn more.

Optimization for convex functions is simpler than for ordinary functions because we don't have to worry about local optima - any stationary point (point where the gradient is zero) is a global minimum. A set S in $\Re^p$ is convex if any line segment between two points in S lies entirely within