Root finding algorithms

Consider a function $f(x)$ for $x \in (a, b)$. A *root* of f is a number $x^\star$ such that $f(x^\star) = 0$. Many computational problems can be viewed as a root finding problem, in particular optimums of a function correspond to the roots of the derivative.

1 Bisection

Let a *bracket* for f be an interval (a, b) such that either $f(a) < 0, f(b) > 0$ or $f(a) > 0, f(b) < 0$. So a bracket is an interval over which the function is guaranteed to have changed sign. By the intermediate value theorem (if f is continuous on (a, b)), a bracket must contain a root of f.

The bisection method requires you to provide a bracket, $c(a_0, b_0)$ as a starting point. The method proceeds by splitting the interval in half, and throws out the interval where the sign did not change, and repeats. Once the interval shrinks below a certain length, δ, it is considered to have converged. Put formally, at step k,

- Calculate $m = (a_{k-1} + b_{k-1})/2$, the midpoint of the current bracket
- if $f(a_{k-1}) \cdot f(m) < 0$ (i.e. the sign does changes over the interval (a_{k-1}, m)), then $a_k = a_{k-1}, b_k = m$.
- otherwise $a_k = m, b_k = b_{k-1}$
- Repeat until $b_k - a_k < \delta$ for some small δ chosen beforehand

At the end, $f(m) \approx 0$. We can see how this works with a plot using the objective function $f(x) = 1/2 - e^{-x^2}$:

```
f <- function(x) .5-exp(-(x^2))
v <- seq(0,2,length=1000)
# function to plot current bracket
plt <- function(f,a0,b0)
{
   plot(v,f(v),type="l", ylim=c(-.5,.5), xlim=c(0,2),
   main=sprintf("Interval length = %f", (b0-a0)))
   abline(h=0)
   segments(a0,.1,a0,-.1,col=2,lty=1)
   points(a0,.1,col=2)
   points(a0,-.1,col=2)
   segments(b0,.1,b0,-.1,col=2,lty=1)
   points(b0,.1,col=2)
   points(b0,-.1,col=2)
   segments(mean(c(a0,b0)), .1, mean(c(a0,b0)), -.1, col=4, lty=1)
   points(mean(c(a0,b0)),.1,col=4)
   points(mean(c(a0,b0)),-.1,col=4)
}

# do one iteration of bisection
```

```
iter1 <- function(f,I)
{
   m = mean(I)
   if( f(I[1])*f(m) < 0 ) return( c(I[1],m) ) else return( c(m, I[2]) )
}

# start values
I <- c(0,2)
f <- function(x) .5-exp(-(x^2))

# re type this several times to see bisection work
a0 <- I[1]; b0 <- I[2];
plt(f, a0, b0)
I = iter1(f,I)
```

We can now write a totally generic function to bisect f to find the root inside some bracket (a,b):

```
# f is the function
# (a,b) is the bracket
# the smaller tol is, the more accurate
bisect <- function(f, a, b, tol)
{


   # initial bracket
   I <- c(a,b)

   # length of current interval
   L <- I[2]-I[1]

   while( L > tol )
   {

      # midpoint
      m <- mean(I)

      # check if f switches signs in the first interal
      if( f(m)*f(I[1]) < 0 ) I = c(I[1],m) else I = c(m,I[2])

      L <- I[2]-I[1]

   }

   return(mean(I))

}
```

```
fn <- function(x) .5-exp(-(x^2))
bisect(fn,0,2,1e-6)
[1] 0.8325543
> fn(.8325543)
[1] -2.590558e-07
```

Notice bisection splits the interval in half each time. Therefore, it will take at L/δ number of iterations before convergence where L is the initial length of the bracket.

Example: Let $X_1, ..., X_n \sim N(\theta, 1)$, where θ is unknown. It is easy to show that the log-likelihood is (up to additive constant):

$$\ell(\theta) = -\frac{1}{2}\sum_{i=1}^{n}(x_i - \theta)^2$$

So the derivative of the log-likelihood (also known as the *score function*) is

$$\ell'(\theta) = \sum_{i=1}^{n}(x_i - \theta)$$

Clearly a root of the score function is $\theta = \overline{X}$, but we will calculate the root using bisection:

```
X <- rnorm(100)
fn <- function(t) sum(X-t)

# apparently (-1,1) is a bracket
c( fn(-1), fn(1) )
[1] 118.60454 -81.39546

bisect(fn, -1, 1, 1e-7)
[1] 0.01425120

mean(X)
[1] 0.01425122
```

This shows the score function has a root at $\overline{X}$, so the log-likelihood has an extrema there. By checking the derivative of the score function, it is clear that this extrema is a maximum.

2 Newton's method

2.1 One-dimensional

Newton's method (or Newton-Raphson method) is based, beginning with some arbitrary x_0, calculate the equation of the line tangent to f at x_0, and see where that tangent line crosses 0. The equation of the line tangent (also the two term Taylor approximation) to f at x_0 is

$$h(x) = f(x_0) + f'(x_0)(x - x_0)$$

The tangent line crosses 0 when $h(x) = 0$, which happens when

$$f(x_0) = -f'(x_0)(x - x_0)$$

rearranging terms approporiately, we find that

$$x = x_0 - f(x_0)/f'(x_0)$$

Now setting $x_0 = x$ and repeating until convergence is essentially Newton's method. We can see graphically how this works with the same test function as before: $f(x) = 1/2 - e^{-x^2}$.

```
# f and f'
f <- function(x) .5-exp(-(x^2))
df <- function(x) 2*x*exp(-(x^2))

# make the initial plot
v <- seq(0,2,length=1000)
plot(v,f(v),type="l",col=8)
abline(h=0)

# start point
x0 <- 1.5

# paste this repeatedly to see Newton's method work
segments(x0,0,x0,f(x0),col=2,lty=3)
slope <- df(x0)
g <- function(x) f(x0) + slope*(x - x0)
v = seq(x0 - f(x0)/df(x0),x0,length=1000)
lines(v,g(v),lty=3,col=4)
x0 <- x0 - f(x0)/df(x0)
```

This is the basic Newton's method. Notice that when we want to optimize a function f, we must find a root of f' (assuming the optimum is not on the boundary of the space of interest, etc.). This suggests optimizing f by starting with some arbitrary point x_0 and updating according to

$$x = x_0 - f'(x)/f''(x)$$

Since we will be using Newton's method in the context of optimization, this will be the algorithm we're referring to. This can also be derived by fitting a local quadratic polynomial to your function at the point x_0 by use of Taylor's Theorem:

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)$$

This local quadratic approximation has an extrema at $x_0 - f'(x_0)/f''(x_0)$. If the target function is exactly quadratic, then Newton's method converges in one step.

Sometimes making a full step of length $f(x)/f'(x)$ is too far and causes us to overshoot the target and actually move farther away from the root. For that reason, we use a slightly modified version of Newton's method in this class. We also include a condition to make sure that the algorithm will converge to a maximum. The formal algorithm, starting from some point x_0 and pre-chosen δ, at step $k \geq 1$:

1. Let $\lambda = 1$
2. $x_k = x_{k-1} - \lambda f'(x_{k-1})/f''(x_{k-1})$
3. if $f(x_k) > f(x_{k-1})$, you have found an uphill step– move on to step 4. Otherwise, let $\lambda = \lambda/2$ and return to step 2
4. if $|f'(x_k)| < \delta$ (the derivative is very close to 0) **or** $|x_k - x_{k-1}|/(\delta + |x_k|) \leq \delta$ (the successive points are very close together), then the algorithm is considered to be converged. If not, move on to the next iteration.

Notice the $f(x_k) > f(x_{k-1})$ also tacitly makes sure the algorithm is looking for a maximum (it would actually be impossible to converge to a minimum using the above algorithm).

2.2 Multi-dimensional

When f is a function of d inputs, the Newton-Raphson algorithm changes very little. The role of f' is now assumed by the gradient

$$\nabla f = (\partial f/\partial x_1, ..., \partial f/\partial x_d)$$

and the role of the second derivative is assumed by the *hessian matrix*, $\nabla^2 f$ which is the matrix of second derivatives:

$$(\nabla^2 f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

The absolute values become norms. If $\mathbf{x} = (x_1, ...x_d)$, then we will take the norm of $\mathbf{x}$ to be

$$||\mathbf{x}|| = \sqrt{\sum_{k=1}^{d} x_k^2}$$

which is called the euclidean norm. So, now the algorithm becomes:

1. Let $\lambda = 1$
2. $x_k = x_{k-1} - \lambda \left(\nabla^2 f(x_{k-1}) \right)^{-1} \nabla f(x_{k-1})$
3. if $f(x_k) > f(x_{k-1})$, you have found an uphill step– move on to step 4. Otherwise, let $\lambda = \lambda/2$ and return to step 2
4. if $||\nabla f(x_{k-1})|| < \delta$ (the gradient norm is small) **or** $||x_k - x_{k-1}||/(\delta + ||x_k||) \leq \delta$ (the successive points are very close together), then the algorithm is considered to be converged. If not, move on to the next iteration.

We can write this as a generic function in R:

```
# f: the function you want the max of
# df: derivative of f; d2f: second derivative
# x0: start value, tol: convergence criterion
# maxit: max # of iterations before it is considered to have failed
newton <- function(x0, f, df, d2f, tol=1e-4, pr=FALSE)
{

   # iteration counter
   k <- 1

   # initial function, derivatives, and x values
   fval <- f(x0)
   grad <- df(x0)
   hess <- d2f(x0)
   xk_1 <- x0

   cond1 <- sqrt( sum(grad^2) )
   cond2 <- Inf

   # see if the starting value  is already close enough
   if( (cond1 < tol) ) return(x0)

   while( (cond1 > tol) & (cond2 > tol) )
   {

      L <- 1
      bool <- TRUE
      while(bool == TRUE)
      {

         xk <- xk_1 - L * solve(hess) %*% grad

         # see if we've found an uphill step
         if( f(xk) > fval )
         {

            bool = FALSE
            grad <- df(xk)
            fval <- f(xk)
            hess <- d2f(xk)
         # make the stepsize a little smaller
         } else
         {
           L = L/2
           if( abs(L) < 1e-20 ) return("Failed to find uphill step - try new start val
         }

      }
```

```
        # calculate convergence criteria
        cond1 <- sqrt( sum(grad^2) )
        cond2 <- sqrt( sum( (xk-xk_1)^2 ))/(tol + sqrt(sum(xk^2)))

        # add to counter and update x
        k <- k + 1
        xk_1 <- xk


     }

     if(pr == TRUE) print( sprintf("Took %i iterations", k) )
     return(xk)

}
```

Examples:

Example 1: Let $f(x) = e^{-(x^2)}$. It is easy to see that $f'(x) = -2xf(x)$ and $f''(x) = -2f(x) - 2xf'(x)$. Now we code each of these functions and call the `newton()` function:

```
f <- function(x) exp(-(x^2))
df <- function(x) -2*x*f(x)
d2f <- function(x) -2*f(x)-2*x*df(x)

# Newton's method starting at x0 = 2/3
newton(2/3, f, df, d2f, 1e-7)
              [,1]
[1,] -3.233794e-09
```

The maximum at 0 appears to have been found. Bad starting values will cause the algorithm to fail, though.

Example 2: Suppose $X_1, ..., X_n$ are iid $N(\mu, \sigma^2)$ where both μ and σ^2 are unknown. We will write a program to do maximum likelihood estimation for $\theta = (\mu, \sigma^2)$. The log-likelihood is

$$\ell(\mu, \sigma^2) = -\frac{1}{2}\left(n\log(\sigma^2) + \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2\right)$$

so the elements of the gradient are

$$\frac{\partial \ell}{\partial \mu} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)$$

and

$$\frac{\partial \ell}{\partial \sigma^2} = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^{n}(x_i - \mu)^2$$

finally, the elements of the hessian are

$$\frac{\partial^2 \ell}{\partial \mu^2} = -\frac{n}{\sigma^2},$$

$$\frac{\partial^2 \ell}{\partial (\sigma^2)^2} = \frac{n}{2\sigma^4} - \frac{1}{\sigma^6}\sum_{i=1}^{n}(x_i - \mu)^2$$

and

$$\frac{\partial^2 \ell}{\partial \mu \partial \sigma^2} = -\frac{1}{\sigma^4}\sum_{i=1}^{n}(x_i - \mu)$$

We can code these functions easily in R and call `newton` to get the MLEs:

```
X <- rnorm(100, mean=2, sd=sqrt(4))
n <- 100
# likelihood t[1]: mu, t[2]: sigma^2
# if sigma < 0 return -Inf, instead of NaN.
f <- function(t)
{
    if( t[2] > 0)
    {
       return( sum( dnorm(X, mean=t[1], sd=sqrt(t[2]), log=TRUE) ) )
    } else
    {
       return(-Inf)
    }
}

# score function
df <- function(t)
{
 mu <- t[1]; sig <- t[2];
 g <- rep(0,2)
 g[1] <- (1/sig) * sum(X - mu)
 g[2] <- (-n/(2*sig)) + sum( (X-mu)^2 )/(2*sig^2)
 return(g)
}

# hessian
d2f <- function(t)
{
 mu <- t[1]; sig <- t[2];
 h <- matrix(0,2,2)
 h[1,1] <- -n/sig
 h[2,2] <- (n/(2*sig^2)) - sum( (X-mu)^2 )/(sig^3)
 h[1,2] <- -sum( (X-mu) )/(sig^2)
 h[2,1] <- h[1,2]
 return(h)
}

newton( c(0,1), f, df, d2f)
         [,1]
[1,] 1.756834
[2,] 5.398555

c( mean(X), (n-1)*var(X)/n)
[1] 1.756834 5.398559
```

We know the true MLEs are $\overline{X}$ and $\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{X})^2$, which is about what the algorithm converged to.

Example 3: In the last example we know the MLEs analytically. In this case the solution must be calculated numerically. Let $X_1, ..., X_n \sim \text{Gamma}(\alpha,\beta)$. That is, each X_i has density

$$\frac{1}{\Gamma(\alpha)\beta^{\alpha}}x^{\alpha-1}e^{-x/\beta}$$

So the log-likelihood for the entire sample is

$$\ell(\alpha,\beta) = -n\left(\log(\Gamma(\alpha)) + \alpha\log(\beta)\right) - \frac{1}{\beta}\sum_{i=1}^{n}X_i + (\alpha-1)\sum_{i=1}^{n}\log(X_i)$$

The elements of the gradient are

$$\frac{\partial\ell}{\partial\alpha} = -n\psi(\alpha) - n\log(\beta) + \sum_{i=1}^{n}\log(X_i)$$

and

$$\frac{\partial\ell}{\partial\beta} = -\frac{n\alpha}{\beta} + \frac{1}{\beta^2}\sum_{i=1}^{n}X_i$$

where ψ is the digamma function, calculable in R with the function `digamma`. The elements of the hessian matrix are:

$$\frac{\partial^2\ell}{\partial\alpha^2} = -n\varphi(\alpha)$$

$$\frac{\partial^2\ell}{\partial\beta^2} = \frac{n\alpha}{\beta^2} - \frac{2}{\beta^3}\sum_{i=1}^{n}X_i$$

and

$$\frac{\partial^2\ell}{\partial\alpha\partial\beta} = -\frac{n}{\beta}$$

where φ is the trigamma function, calculable with the R function `trigamma`. We can code these into R and use `newton` to calculate the MLE for $\theta = (\alpha,\beta)$.

```
n <- 100
X <- rgamma(n, shape=2, scale=3)

# t[1]: alpha, t[2]: beta
f <- function(t)
{
    a <- t[1]; b <- t[2];
    if( (a > 0) & (b > 0) )
    {
       return( sum( dgamma(X, a, scale=b, log=TRUE) ) )
    } else
    {
       return(-Inf)
    }
}

df <- function(t)
{
   g=c(0,0)
   a <- t[1]; b <- t[2];
   if( (a > 0) & (b > 0) )
   {
      g[1] <- -n*digamma(a) - n*log(b) + sum(log(X))
      g[2] <- -n*a/b + sum(X)/(b^2)
   } else
   {
     g <- c(-Inf, -Inf)
   }
  return(g)
}
d2f <- function(t)
{
   h = matrix(0,2,2)
   a <- t[1]; b <- t[2];
   if( (a>0) & (b>0) )
   {
      h[1,1] = -n*trigamma(a)
      h[2,2] = n*a/(b^2) - 2*sum(X)/(b^3)
      h[1,2] = -n/b
      h[2,1] = -n/b
   } else
   {
     h <- matrix(-Inf, 2, 2)
   }
 return(h)
}

newton( c(1,5), f, df, d2f)
```

```
         [,1]
[1,] 2.095618
[2,] 3.056548

# try larger sample size
n <- 1e5
X = rgamma(n, shape=2, scale=3)
newton( c(1,5), f, df, d2f)
         [,1]
[1,] 2.004633
[2,] 3.003172
```

From the asymptotic unbiasedness it looks like we have found the MLEs, although a better check would be to see that the asymptotic covariance matrix is equal to the inverse of the fisher information for a single observation.

Example 4: As a higher dimensional example consider an ordinary regression model

$$Y_i = \beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p} + \varepsilon_i$$

where $\varepsilon_i \sim N(0,1)$. The predictors are viewed as fixed constants. We want to estimate β by maximum likelihood using newton raphson. Since $Y_i \sim N(\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p}, 1)$, the log-likelihood for subject i is

$$\ell_i(\beta) = -\frac{1}{2}(Y_i - (\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p}))^2$$

So, the gradient with respect to β_j is

$$\frac{\partial \ell_i}{\partial \beta_j} = X_{i,j}(Y_i - (\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p}))$$

Therefore, the entries of the hessian have the form

$$\frac{\partial^2 \ell_i}{\partial \beta_j \partial \beta_k} = -X_{i,j} X_{i,k}$$

The log-likelihood, score, and hessian of the entire sample can be found by summing the terms for each subject. We code each of these functions into R and calculate the MLE $\hat{\beta}$ using Newton's method:

```
# Generate some fake data with
# an intercept and 4 predictors
n <- 100
Beta <- c(.5, 1.2, -1.8, 3.6, 2.1)
X <- cbind( rep(1,n), rnorm(n), rnorm(n), rnorm(n), rnorm(n) )
Y <- rep(0,n)
for(i in 1:n) Y[i] <- sum( Beta * X[i,] ) +  rnorm(1)

p <- ncol(X)

# full sample log-likelihood
```

```
L <- function(B)
{
   like <- 0
   for(i in 1:n)
   {

      # subject i's mean
      mu_i <- sum( X[i,] * B )

      like <- like + dnorm(Y[i], mean=mu_i, sd=1, log=TRUE)

   }

   return(like)

}

# full sample score function
dL <- function(B)
{
    grad <- rep(0, p)
    for(i in 1:n)
    {

       mu_i <- sum( X[i,] * B )

       for(j in 1:p)
       {

          grad[j] <- grad[j] + X[i,j]*(Y[i] - mu_i)

       }

    }

    return(grad)

}

# full sample hessian matrix
d2L <- function(B)
{

   H <- matrix(0, p, p)
   for(i in 1:n)
   {

      for(j in 1:p)
      {

         for(k in 1:p)
         {

            H[j,k] <- H[j,k] - X[i,j]*X[i,k]

         }

      }

   }

   return(H)

}

### Test on our fake data and compare with lm() output
newton(rep(0,p), L, dL, d2L)
          [,1]
[1,]  0.726647
[2,]  1.034866
[3,] -1.670557
[4,]  3.646216
[5,]  2.221464

lm(Y ~ X[,2] + X[,3] + X[,4] + X[,5])$coef
(Intercept)      X[, 2]      X[, 3]      X[, 4]      X[, 5]
   0.726647    1.034866   -1.670557    3.646216    2.221464
```

So, even starting from completely naive starting values (all 0's) we converge to the true MLE (since the least squares estimator is equivalent to the MLE when the errors are normally distributed).