Lecture 11: 数值优化方法

张伟平

Monday 23^{rd} November, 2009

Contents

Chapter 1

Numerical optimization methods in R

1.1 Root-finding in one dimension

假设$f : R \to R$为一连续函数, 则方程$f(x) = c$的根x, 满足$g(x) = f(x) - c = 0$. 因此我们 只考虑$f(x) = 0$形式的方程求根问题. 使用数值方法求此方程的根, 可以选择是使用f的一阶导数还是 不使用导数的方法. Newton方法或者Newton-Raphson方法是使用一阶导数的方法, 而Brent的最小化 算法[1]是不使用导数的一种求根方法. 在R中, 函数**uniroot**就是基于Brent的求根算法. 该算法的Fortran 程序源代码可以在下面网址上找到http://www.gnu.org/software/gsl/.

[1] R. Brent. Algorithms for minimization without derivatives. Prentice-Hall, New Jersey, 1973

1.1.1 Bisection method

如果$f(x)$在区间$[a,b]$上连续, 以及$f(a)$和$f(b)$有相反的符号, 则由中值定理知道 存在$a<c<b$, 使得$f(c)=0$. 二分法通过在每次迭代中简单的判断$f(x)$在中点$x=(a+b)/2$处的符号来寻求 方程的根. 如果$f(a)$和$f(x)$有相反的符号, 则区间就被$[a,x]$替代, 否则就被$[x,b]$替代. 在每次迭代中, 包含根的区间长度减少一半. 即

1. 记$a_0=a, b_0=b$, 以及$x^{(0)}=(a+b)/2$.
2. 更新区间为

$$[a_{t+1}, b_{t+1}] = \begin{cases} [a_t, x^{(t)}], & f(a_t)f(x^{(t)}) \le 0 \\ [x^{(t)}, b_t], & f(a_t)f(x^{(t)}) > 0 \end{cases}$$

以及 $x^{(t+1)}=(a_{t+1}+b_{t+1})/2$.

3. 循环2直至达到收敛准则.

常用的收敛准则有

绝对收敛

$$|x^{(t+1)} - x^{(t)}| < \epsilon$$

其中ϵ是选定的可容忍的精度. 对于二分法, 可以验证

$$b_t - a_t = 2^{-t}(b_0 - a_0)$$

因此, 当$2^{-(t+1)}(b_0 - a_0) < \delta$时, 即$t > log_2\{(b_0 - a_0)/\delta\} - 1$, 将达到容忍精度$|x^{(t)} - x^*| < \delta$.

可以看出, 二分法不会失效, 达到指定精度所需要的迭代次数也是事先可以得到的. 如果在区间$[a, b]$里 方程有多个根, 则二分法会找到一个根. 二分法的收敛速度是线性的.

相对收敛

相对收敛准则要求

$$\frac{|x^{(t+1)} - x^{(t)}|}{|x^{(t)}|} < \epsilon$$

时停止迭代. 此准则可以不考虑x的单位的情况下达到指定的精度.

例1 解方程

$$a^2+y^2+\frac{2ay}{n-1}=n-2$$

其中a为常数, $n>2$为一整数. 显然, 方程的解为

$$y=-\frac{a}{n-1}\pm\sqrt{n-2+a^2+(\frac{a}{n-1})^2}$$

下面我们使用二分法求此方程的一个数值解. 我们首先要找 一个区间, 比如$(0,5n)$, 使得函数$f(y)=a^2+y^2+\frac{2ay}{n-1}-n+2$在区间两端有着不同的符号. 然后即可以使用二分法.

↑Code

```
a <- 0.5
n <- 20
cat("true roots",-a/(n-1)-sqrt(n-2-a^2+(a/(n-1))^2),
            -a/(n-1)+sqrt(n-2-a^2+(a/(n-1))^2),"\n")

bisec<-function(b0,b1){
```

```
 f <- function(y, a, n) {
    a^2 + y^2 + 2*a*y/(n-1) - (n-2)
}
#solve using bisection
it <- 0
eps <- .Machine$double.eps^0.25
r <- seq(b0, b1, length=3)
y <- c(f(r[1], a, n), f(r[2], a, n), f(r[3], a, n))
if (y[1] * y[3] > 0)
    stop("f does not have opposite sign at endpoints")
while(it < 1000 && abs(y[2]) > eps) {
    it <- it + 1
    if (y[1]*y[2] < 0) {
        r[3] <- r[2]
        y[3] <- y[2]
    } else {
        r[1] <- r[2]
        y[1] <- y[2]
    }
    r[2] <- (r[1] + r[3]) / 2
    y[2] <- f(r[2], a=a, n=n)
```

```
    print(c(r[1], y[1], y[3]-y[2]))
  }
}
bisec(0,5*n)
```

↓Code

精确的解为$y = 4.186841, -4.239473$.

1.1.2 Brent's method

二分法是一种特殊的插入根算法. Brent 通过逆二次插值方法将插入根方法和二分法结合起来. 其使用y的二次函数来拟合x. 如果三个点为$(a, f(a)), (b, f(b)), (c, f(c))$, 其中$b$ 为当前最好的估计, 则通过Lagrange多项式插值方法$(y = 0)$对方程的根进行估计,

$$x = \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]}$$

$$+\frac{[y-f(c)][y-f(a)]b}{[f(b)-f(c)][f(c)-f(a)]}$$

在R中, 函数**uniroot**就是应用Brent方法求解一元方程的数值根.

例2 应用*uniroot*求例1中方程的根.

↑Code

```
a <- 0.5
n <- 20
out <- uniroot(function(y) {
           a^2 + y^2 + 2*a*y/(n-1) - (n-2) },
           lower = 0, upper = n*5)
unlist(out)
uniroot(function(y) {a^2 + y^2 + 2*a*y/(n-1) - (n-2)},
        interval = c(-n*5, 0))$root
```

↓Code

函数*polyroot*求一个系数为复数的多项式的根. 因此, 此例的问题使用*polyroot*函数为

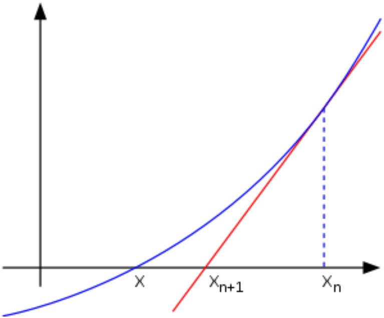
```
polyroot(c(a^2-(n-2),2*a/(n-1),1))
```

↑Code

↓Code

1.1.3 Newton's method

Newton方法是一种快速求根的方法, 有时也称为Newton-Raphson迭代算法. 假设$f(x)$是连续且可微的, 且$f'(x) \neq 0$. 假设我们想要求根x, 我们已经有了当前的一个近似值x_n, 则由下图可以看出, 通过导数表示斜率这一特点, 可以给出x的一个更好的近似.



即

$$f'(x_n) = \frac{f(x_n) - 0}{x_n - x_{n+1}}$$

于是, 解上述方程得到

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

Newton 方法还可以用于最小化或者最大化一个目标函数，由于目标函数g在极值点处的导数为零，因此最 小化或者最大化等价于求方程根:

$$g'(x) = 0$$

从而迭代求解为

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

例 3 使用Newton方法求例1方程的根.

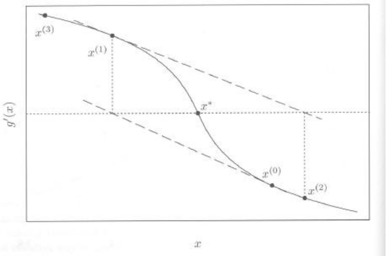
↑Code

```
nt<-function(b0){
  a <- 0.5
  n <- 20
  f <- function(y, a, n) {
      a^2 + y^2 + 2*a*y/(n-1) - (n-2)
      }
  fd<-function(y,a,n){
      2*y+2*a/(n-1)
      }
```

```
 b1<-b0
 b0<-b0-1
 eps <- .Machine$double.eps^0.25
 it<-0
 while(it<1000 && abs(b1-b0)>eps){
 it<-it+1
 b0<-b1
 b1<-b0-f(b0,a,n)/fd(b0,a,n)
 cat(it,c(b0,b1,abs(b1-b0)),"\n")
 }
}
```

↓Code

Newton 方法依赖于f的形状和初值. 下图的例子说明Newton方法从初值开始就发散.



我们下面分析一下相邻两步之间的差异. 假设f的二阶导数存在连续,且$f'(x^*) \neq 0$. 因为 $f'(x^*) \neq 0$且f'在x^*处连续, 则必存在x^*的一个邻域, 使得在此区域里, $f'(x) \neq 0$. 定义$\epsilon_t = x_t - x^*$. 以下我们仅在此邻域内考虑.

由Taylor展开式有

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

其中ξ位于x_t和x^*之间. 整理得到

$$x_t + h_t - x^* = (x^* - x_t)^2 \frac{f''(\xi)}{2f'(x_t)}$$

其中$h_t = -\frac{f(x_t)}{f'(x_t)}$表示Newton更新增量. 此式左边等于$x_{t+1} - x^*$, 因此有

$$\epsilon_{t+1} = \epsilon_t^2 \frac{f''(\xi)}{2f'(x_t)}$$

现对某个$\delta > 0$, 考虑x^*的邻域$\mathcal{N}_\delta(x^*) = [x^* - \delta, x^* + \delta]$, 记

$$c(\delta) = \max_{x_1, x_2 \in \mathcal{N}_\delta(x^*)} \left|\frac{f''(\xi)}{2f'(x_t)}\right|$$

由于当$\delta \to 0$时, $c(\delta) \to |\frac{f''(x^*)}{2f'(x^*)}|$, 所以当$\delta \to 0$时, $\delta \times c(\delta) \to 0$. 因此选择$\delta$,使得$\delta \times c(\delta) < 1$, 于是

$$|c(\delta)\epsilon_{t+1}| = |c(\delta)\epsilon_t^2 \frac{f''(\xi)}{2f'(x_t)}| \leq (c(\delta)\epsilon_t)^2$$

假设一个初值满足$|\epsilon_0| = |x_0 - x^*| \leq \delta$, 则有

$$|\epsilon_t| \leq \frac{(c(\delta)\delta)^{2t}}{c(\delta)}$$

因此, 当$t \to \infty$时, 上式趋于0.

从而我们证明了如下结论

Theorem 1. 若函数f二阶可微连续, 且x^*为方程$f(x) = 0$的一个单根, 则存在x^* 的一个邻域, 当初值为此邻域内任一点时, *Newton*方法得到的解都收敛到x^*.

另外, 当f为二阶可微连续, 为凸函数且根存在, 则无论初值如何取, Newton方法都收敛到此根.

称某个算法的收敛阶数为β, 如果$\lim_{t\to\infty} \epsilon_t = 0$且

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

其中$c \neq 0$且$\beta > 0$.

对于Newton方法, 由于

$$\frac{\epsilon_{t+1}}{\epsilon_t^2} = \frac{f''(\xi)}{2f'(x_t)}$$

因此Newton方法是二次收敛的.

Newton 上山法 (Newton downhill method)

在求方程$f(x) = 0$的根的过程中, 为了避免发散或者迭代次数的增加这些情况, 可以再附加一个条件,

$$f(x_{n+1}) < f(x_n)$$

以保证函数单调下降. 将此条件和Newton方法结合起来, 即称为Newton下山法. 此时迭代方程可以取为

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

其中$0 < \lambda <= 1$称为下山因子. λ的取值是逐步试探的, 首先选择$\lambda = 1$, 然后将λ逐步减半进行试算. 若附加条件满足, 则下山成功; 否则, 下山失败, 需要另选初值再试.

例4 求方程$x^3 - x - 1 = 0$在$x = 1.5$附近的一个根.

若初值取为$x_0 = 0.6$, 则使用Newton方法第一次迭代得到$x_1 = 17.9$大大偏离了方程在1.5附近的根. 因此 迭代过程可能发散或者迭代次数增加. 若使用Newton下山法, 选择$\lambda = 1/32$, 则$x_1 = 1.40625$, 比较 靠近方程的根.

正切法(Secant method)

在Newton算法中, 需要计算f的导数, 如果f的导数很难计算, 则可以使用离散差分来近似之:

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

称此方法为正切法. 此方法需要两个初值x_0, x_1. 可以证明, 正切法的收敛速度要低于Newton方法.

1.1.4 Fisher scoring

在求解MLE问题里, 此时等价于求方程

$$f'(\theta) = 0$$

的根. f'表示对数似然函数的导数. 因此使用Newton方法迭代方程为

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

注意到此时$-1/f''(\theta) = I(\theta)$, 因此使用Fisher信息量, 迭代方程为

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

称此方法为Fisher得分法.

Fisher得分法和Newton法具有相同的渐近性质, 但对于个别问题, 一个可能比另一个易于计算或分析. 一般来讲, Fisher得分法在迭代之初效果明显, 而Newton方法则在迭代结束前效果明显.

1.2 multivariate optimization

在目标函数为多元时, 如$f(\mathbf{x}) = f(x_1, \cdots, x_p)$, 令$\mathbf{x}^{(t)} = (x_1^{(t)}, \cdots, x_p^{(t)})$ 表示在第t步迭代后的估计. 则前面提到的方法一般也是适用于这种场合的. 收敛准则 也可以类似的取为

$$D(u, v) = \sum_{i=1}^{p} |u_i - v_i|, \quad \text{或者} \quad D(u, v) = \sqrt{\sum_{i=1}^{p} (u_i - v_i)^2}$$

于是绝对收敛准则和相对收敛准则分别为

$$D(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}) < \epsilon$$

和

$$\frac{D(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)})}{D(\mathbf{x}^{(t)}, 0)} < \epsilon$$

1.2.1 Newton's method and Fisher scoring

我们对$f(x^*)$作二阶Taylor逼近

$$f(x^*) = f(\mathbf{x}^{(t)}) + (x^* - \mathbf{x}^{(t)})^T f'(\mathbf{x}^{(t)}) + \frac{1}{2}(x^* - \mathbf{x}^{(t)})^T f''(\mathbf{x}^{(t)})(x^* - \mathbf{x}^{(t)})$$

然后关于x^*最大化此逼近, 以得到下一步的更新值. 令上式的梯度为0,得到

$$f'(\mathbf{x}^{(t)}) + f''(\mathbf{x}^{(t)})(x^* - \mathbf{x}^{(t)}) = 0$$

从而得到更新方程

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - [f''(\mathbf{x}^{(t)})]^{-1} f'(\mathbf{x}^{(t)})$$

如果使用一个矩阵$M^{(t)}$来近似Hessian阵, 则

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - [M^{(t)})]^{-1} f'(\mathbf{x}^{(t)})$$

此类方法称为Quasi-Newton方法.

同单变量情形, 在求解MLE问题中, 我们可以使用Fisher信息阵在$\theta^{(t)}$处的值 来替代Hessian阵, 即

$$\theta^{(t+1)} = \theta^{(t)} + I^{-1}(\theta^{(t)})f'(\theta^{(t)})$$

这里f表示对数似然函数.

1.3 Numerical Integration

数值积分方法可以是自适应的或者非自适应的. 数值积分方法 总是通过一个有限点集上函数值的加权和的形式来估计积分值,

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} f(x_i)w_i$$

其中w_i为权重. 选取这些点集和权重的不同方法, 会导致估计积分的精度不同. 因此, 总是 可以通过积分精度来评价不同的数值积分方法. 非自适应的方法在积分区域上使用同一个准则. 很多非自适应 的数值积分方法都是通过建立插值函数来估计积分. 常见的有

矩形法则(Rectangle rule)

最简单的插值函数就是常数函数, 其通过点$(\frac{a+b}{2}, f(\frac{a+b}{2}))$, 因此

$$\int_a^b f(x)dx \approx (b-a)f(\frac{a+b}{2})$$

梯形法则(Trapezoidal rule)

插值函数可以是一个仿射函数(affine function, 1次多项式), 其通过点$(a, f(a))$和$(b, f(b))$, 即

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2}$$

混合的梯形法则(Composite trapezoidal rule)

在积分区间为有限区间时, 将区间$[a, b]$分割为长度相同的n个子区间, 每个长度为$h = (b-a)/n$, 每个区间的端点为$a = x_0, x_1, \cdots, x_n = b$, 使用梯形的面积来近似每个子区间上的积分值, 如在区间(x_i, x_{i+1}) 上, 梯形面积

为$(f(x_i)+f(x_{i+1}))h/2$. 因此整个区间上积分值估计为

$$\frac{h}{2}f(a)+h\sum_{i=1}^{n-1}f(x_i)+\frac{h}{2}f(b)$$

Newton-Cotes 公式

将积分区间分割为等间距的n个子区间, 得到点集$(x_0,f(x_0))$, $(x_1,f(x_1))$, $\cdots$, $(x_n,f(x_n))$, 然后通过这$n+1$个点构造n次多项式$L(x)$.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} f(x_i)\int_a^b l_i(x)dx$$

Gaussian quadrature

如果我们允许积分区间$[a,b]$可以分割为不等间距的区间, 使用得到的$n+1$个点构建n阶多项式来近似被积函数, 则 此时的方法称为高斯积分法.

当被积函数在积分区域的子集上性质良好, 而在其他子集上不是很好时, 那么将积分区域分割为这些单独的子集来考虑 有助于提供估计的精度. 自适应

方法就是通过被积函数在子区间上的性质来选择分割区间.

R中计算一元函数数值积分的函数为**integrate**. 其使用积分自适应方法来估计积分, 而且允许积分区域为无穷.

例 4 计算积分

$$\int_0^{\infty} \frac{dy}{(coshy - \rho r)^{n-1}}$$

其中$-1 < \rho, r < 1, n \geq 2$为整数.

我们使用R中的函数**integrate**来计算此积分. 对固定的参数, 比如$n = 10, r = 0.5, \rho = 0.2$ 积分积分值是很方便的, 如

```
integrate(function(y){(cosh(y)-0.1)^(-0.9)},0,Inf)
```

↑Code ↓Code

对任意的参数值, 我们需要使用带参数的被积函数

```
f <- function(y, N, r, rho) {(cosh(y) - rho * r)^(1 - N) }
```

↑Code

```
integrate(f, lower=0, upper=Inf, rel.tol=.Machine$double.eps^0.25,
          N=10, r=0.5, rho=0.2)
```

↓Code

积分值依赖参数ρ的情况, 可以通过图示来观察(下面固定$n = 10, r = 0.5$)

↑Code

```
ro <- seq(-.99, .99, .01)
v <- rep(0, length(ro))
for (i in 1:length(ro)) {
    v[i] <- integrate(f, lower=0, upper=Inf,
              rel.tol=.Machine$double.eps^0.25,
              N=10, r=0.5, rho=ro[i])$value
    }
plot(ro, v, type="l", xlab=expression(rho),
     ylab="Integral Value (n=10, r=0.5)")
```

↓Code

例 5, 样本相关系数密度 样本相关系数

$$R = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{[\sum_{i=1}^n (X_i - \bar{X})^2 \sum_{j=1}^n (Y_j - \bar{Y})^2]^{1/2}}$$

记ρ为总体相关系数. 则在$(X_1, Y_1), \cdots, (X_n, Y_n), i.i.dN(\mu_1, \mu_2, \sigma_1^2, \sigma_2^2, \rho)$假设下, 以及$\rho = 0$时, 可以证明$R$的密度为

$$f(r) = \frac{\Gamma((n-1)/2)}{\Gamma(1/2)\Gamma((n-2)/2)}(1-r^2)^{(n-4)/2}, \quad -1 < r < 1$$

对$0 < |\rho| < 1$, R的密度很复杂, 一种表示方法为

$$f(r) = \frac{(n-2)(1-\rho^2)^{(n-1)/2}(1-r^2)^{(n-4)/4}}{\pi} \int_0^\infty \frac{dw}{(coshw - \rho r)^{n-1}}$$

其中$-1 < r < 1$.

因此此时如果要计算密度在给定点处的值, 就必须计算其中的积分. 我们使用**integrate**函数来计算密度的值.

↑Code

```
.dcorr <- function(r, N, rho=0) {
        # compute the density function of sample correlation
        if (abs(r) > 1 || abs(rho) > 1) return (0)
        if (N < 4) return (NA)
        if (isTRUE(all.equal(rho, 0.0))) {
            a <- exp(lgamma((N - 1)/2) - lgamma((N - 2)/2)) /
                     sqrt(pi)
            return (a * (1 - r^2)^((N - 4)/2))
        }
        # if rho not 0, need to integrate
        f <- function(w, R, N, rho)
            (cosh(w) - rho * R)^(1 - N)
        #need to insert some error checking here in case
        # the numerical integration fails
        i <- integrate(f, lower=0, upper=Inf,
                R=r, N=N, rho=rho)$value
        c1 <- (N - 2) * (1 - rho^2)^((N - 1)/2)
        c2 <- (1 - r^2)^((N - 4) / 2) / pi
        return(c1 * c2 * i)
    }
```

↓Code

对ρ取$0, 0.5, -0.5$画出密度图:

↑Code

```
r <- as.matrix(seq(-1, 1, .01))
d1 <- apply(r, 1, .dcorr, N=10, rho=.0)
d2 <- apply(r, 1, .dcorr, N=10, rho=.5)
d3 <- apply(r, 1, .dcorr, N=10, rho=-.5)
plot(r, d2, type="l", lty=2, lwd=2, ylab="density")
lines(r, d1, lwd=2)
lines(r, d3, lty=4, lwd=2)
legend("top", inset=.02,
       c("rho = 0", "rho = 0.5", "rho = -0.5"), lty=c(1,2,4), lwd=2)
```

↓Code

1.4 Maximum Likelihood Problems

在R中, 函数**mle**用来求极大似然估计, 其使用函数**optim**来对负对数似然函数进行最小化.

例 6 极大似然估计 假设Y_1, Y_2 $i.i.d$ 参数为θ的指数分布, 求θ的MLE.

此例可以得到θ的MLE为

$$\hat{\theta} = \frac{Y_1 + Y_2}{2}$$

下面我们使用**mle**函数来求MLE. 该函数的第一个参数为负的对数似然, 因此R代码如下

↑Code

```
#the observed sample
   y <- c(0.04304550, 0.50263474)
   mlogL <- function(theta=1) {
       #minus log-likelihood of exp. density, rate 1/theta
       return( - (length(y) * log(theta) - theta * sum(y)))
   }
   library(stats4)
   fit <- mle(mlogL)
   summary(fit)
```

↓Code

另外, 还可以在函数**mle**里指定初始值:

↑Code

```
# Alternately, the initial value for the optimizer could
    # be supplied in the call to mle; two examples are
    mle(mlogL, start=list(theta=1))
    mle(mlogL, start=list(theta=mean(y)))
```

↓Code

1.5 Optimization Problems

1.5.1 One-dimension Optimization

许多优化问题可以转换为寻求一个函数的根, 因此此时**uniroot**函数可以用来解决问题. 函数 **nlm**使用Newton类型算法进行非线性优化, 函数**optimize**是一个面向连续函数优化问题而设计的, 使用黄金分割方法以及曲线插值等算法来最优化目标函数.

例 7 一维优化问题 求下面函数的最大值点

$$f(x) = \frac{log(1 + log(x))}{log(1 + x)}$$

此函数的图像见

↑Code

```
x <- seq(2, 8, .001)
y <- log(x + log(x))/(log(1+x))
plot(x, y, type = "l")
```

↓Code

因此使用**optimize**在区间$(4, 8)$上来优化此函数

↑Code

```
 f <- function(x)
    log(x + log(x))/log(1+x)
optimize(f, lower = 4, upper = 8, maximum = TRUE)
```

↓Code

optimize函数使用黄金分割方法寻求最小值或者最大值点. 目标函数f的第一个评估点一般总是 $x_1 = a + (1 - \phi)(b - a)$, 这里$(a, b) = (lower, upper)$以及$\phi = (sqrt5 - 1)/2 = 0.61803$, 黄金分割比例. 第二个评估点为$x_2 = a + \phi(b - a)$. 然后$[x_1, x_2]$ 内的最小值就会被作为最终的最小值. 因此区间$(lower, upper)$的指定有可能会影响优化的结果.

↑Code

```
## "wrong" solution with unlucky interval and piecewise constant f():
f  <- function(x) ifelse(x > -1, ifelse(x < 4, exp(-1/abs(x - 1)), 10), 10)
fp <- function(x) { print(x); f(x) }
plot(f, -2,5, ylim = 0:1, col = 2)
optimize(fp, c(-4, 20))# doesn't see the minimum
optimize(fp, c(-7, 20))# ok
```

↓Code

例 8 MLE: Gamma分布 假设$x_1, \cdots, x_n$为从$Gamma(r, \lambda)$中抽取的i.i.d样本, 这里r为形状参数, λ为速率参数. 求$\theta = (r, \lambda)$的MLE.

对数似然函数为

$$l(r,\lambda) = nrlog(\lambda) - nlog(\Gamma(r)) + (r-1)\sum_{i=1}^{n} logx_i - \lambda\sum_{i=1}^{n} x_i, x_i \geq 0$$

因此, 最大化此函数是一个二维最优化问题. 但是此问题可以转换为一维优化问题:

$$\frac{\partial}{\partial\lambda} l(r,\lambda) = \frac{nr}{\lambda} - \sum_{i=1}^{n} logx_i = 0;$$

$$\frac{\partial}{\partial r} l(r,\lambda) = nlog\lambda - n\frac{\Gamma'(r)}{\Gamma(r)} + \sum_{i=1}^{n} logx_i = 0$$

由此两式的第一式得到$\hat{\lambda} = \hat{r}/\bar{x}$, 带入到第二式得到

$$nlog\frac{\hat{r}}{\bar{x}} + \sum_{i=1}^{n} logx_i - n\frac{\Gamma'(\hat{r})}{\Gamma(\hat{r})} = 0$$

因此优化问题等价于求此方程的根. 从而MLE就是下述方程的解

$$log\lambda + \frac{1}{n}\sum_{i=1}^{n} logx_i = \psi(\lambda\bar{x}), \quad \bar{x} = \frac{r}{\lambda}$$

其中$\psi(t) = \frac{d}{dt} log\Gamma(t) = \Gamma'(t)/\Gamma(t)$(R中的**digamma**函数). 因此 可以使用**uniroot**来求得数值解. 下面对$r = 5, \lambda = 2$的Gamma分布, 重复$m = 20000$次抽样和 使用*uniroot*函数求解的过程. 最后总结.

↑Code

```
m <- 20000
est <- matrix(0, m, 2)
n <- 200
r <- 5
lambda <- 2

obj <- function(lambda, xbar, logx.bar) {
    digamma(lambda * xbar) - logx.bar - log(lambda)
    }

for (i in 1:m) {
    x <- rgamma(n, shape=r, rate=lambda)
    xbar <- mean(x)
    u <- uniroot(obj, lower = .001, upper = 10e5,
            xbar = xbar, logx.bar = mean(log(x)))
    lambda.hat <- u$root
```

```
        r.hat <- xbar * lambda.hat
        est[i, ] <- c(r.hat, lambda.hat)
    }

    ML <- colMeans(est)
    hist(est[, 1], breaks="scott", freq=FALSE,
        xlab="r", main="")
    points(ML[1], 0, cex=1.5, pch=20)
    hist(est[, 2], breaks="scott", freq=FALSE,
         xlab=bquote(lambda), main="")
    points(ML[2], 0, cex=1.5, pch=20)
```

↓Code

1.5.2 multi-dimensional Optimization

在R中,对多维参数的优化问题, 函数**optim**使用如Nelder-Mead单纯形方法, Quasi-Newton方法, 共轭梯度算法, box constraints优化方法和模拟退火算法等对目标函数进行优化. 语法如下

↑Code

```
optim(par,fn,gr=NULL,method=c("Nelder-Mead","BFGS","CG","L-BFGS-B","SANN"),
lower=-Inf,upper=Inf, control=list(), hessian=FALSE, ...)
```

↓Code

♣ "Nelder-Mead": 单纯形方法, 默认方法, 此方法只使用目标函数的值, 因此很稳健, 但是速度较慢.

♣ "BFGS": 最流行的Quasi-Newton算法之一, 由其作者的名字命名: Broyden, Fletcher, Goldfarb, 和 Shanno. 其使用函数和值以及函数的梯度来最优化目标函数.

♣ "CG": 共轭梯度方法. 此方法一般来讲要比BFGS方法脆弱, 但是由于其在迭代中不需要存储矩阵, 因而 在大维优化问题中有时可以使用.

♣ "L-BFGS-B": 此方法允许给出每个变量的取值范围, 然后使用有限内存拟牛顿方法(limited-memory modification of the BFGS quasi-Newton).

♣ "SANN": 模拟退火(simulated annealing)算法的一种变种. 模拟退火是一类随机全局最优化方法. 其只使用函数的值, 因此相对较慢. SANN方法严重依赖于控制参数.

例 9 使用*optim*求例8的MLE.

↑Code

```
LL <- function(theta, sx, slogx, n) {
        r <- theta[1]
        lambda <- theta[2]
        loglik <- n * r * log(lambda) + (r - 1) * slogx -
            lambda * sx - n * log(gamma(r))
        - loglik
        }
```

↓Code

*optim*函数默认是最小化, 因此我们最优化负的对数似然. 可以使用矩估计方法给出参数的 一个初始值, 这里简单计我们取$r = 1, \lambda = 1$作为初始值.

↑Code

```
n <- 200
r <- 5;     lambda <- 2
x <- rgamma(n, shape=r, rate=lambda)

optim(c(1,1), LL, sx=sum(x), slogx=sum(log(x)), n=n)
```

↓Code

和例8的结果相比较, 我们也重复此优化过程20000次, 得到平均值.

↑Code

```
mlests <- replicate(20000, expr = {
  x <- rgamma(200, shape = 5, rate = 2)
  optim(c(1,1), LL, sx=sum(x), slogx=sum(log(x)), n=n)$par
  })
colMeans(t(mlests))
```

↓Code

1.6 Linear Programming

在R中, 可以使用*boot*包里的**simplex**函数利用单纯形算法 求解一个线性规划问题.

$$
\begin{aligned}
Objective \quad & \min \ c'x \\
& subjectto \\
Constraints \quad & Ax = b; x \geq 0
\end{aligned}
$$

例 10 求解线性规划问题

$$
\begin{aligned}
& \max \ 2x + 2y + 3z \\
\text{subject to :} \quad & -2x + y + z \leq 1 \\
& 4x - y + 3z \leq 3 \\
& x \geq 0; y \geq 0; z \geq 0.
\end{aligned}
$$

使用**simplex**函数求解

↑Code

```
library(boot)   #for simplex function
A1 <- rbind(c(-2, 1, 1), c(4, -1, 3))
b1 <- c(1, 3)
a <- c(2, 2, 3)
simplex(a = a, A1 = A1, b1 = b1, maxi = TRUE)
detach(package:boot)
```

↓Code

除此之外, 还有其他一些函数来求解线性规划问题:

↑Code

```
linp(limSolve)          线性规划
solveLP(linprog)        解线性规划/优化问题
lpcdd(rcdd)             使用精确计算方法求解线性规划
```

↓Code