Numerical Methods

Lecture 7

Nicholas Christian

BIOST 2094 Spring 2011

Outline

Numerical Integration

- The function `integrate()` is used to integrate functions of one variable over a finite or infinite interval
- When integrating over infinite intervals use `-Inf` or `Inf`
- The function you are integrating must accept a vector argument and return a vector with the function evaluated at each point in the argument vector. Use `Vectorize()` to create a function that performs vector computations.
- Using `integrate()` along with `sapply()` allows you to evaluate multiple integrals.

```
integrate(f, lower, upper, ...)
```

`f`	Function integrating
`lower`	Lower limit of integration
`upper`	Upper limit of integration
`...`	Additional arguments passed to `f`
	There are additional arguments for controlling the accuracy of the estimate

Example - Numerical Integration

```
# Simple example
f <- function(x) exp(x)

# Returns a class "integrate" object
(I <- integrate(f, lower=0, upper=1))
names(I)

# Get the estimate of the intergral
integrate(f, lower=0, upper=1)$value

# Integrate a normal density function
integrate(dnorm, -1.96, 1.96, mean=0, sd=1)
integrate(dnorm, -Inf, Inf, mean=0, sd=1)

# Vector Computations
g <- function(x) 1
integrate(g, 0, 1)

# Need to vectorize g
g(1:10)
Vectorize(g)(1:10)
integrate(Vectorize(g), 1, 10)
```

Example - Numerical Multiple Integration

- The following trick came from the R message board

$$\int_0^3 \int_1^2 x^2 y \; dydx$$

```
# Iterated Integral
integrate(function(x) {
     sapply(x, function(x) {
            integrate(function(y) x^2*y, 1, 2)$value
     })
}, 0, 3)
```

$$\int_0^1 \int_x^1 x \sin(y^2) \; dydx$$

```
# Double Integral
integrate(function(x) {
    sapply(x, function(x) {
           integrate(function(y) x*sin(y^2), x, 1)$value
    })
}, 0, 1)
```

Symbolic Derivatives

- To perform symbolic derivatives of simple expressions use `deriv()`

`deriv(expr, namevec, hessian=FALSE)`

`expr`	Either a formula with no left-hand side or an expression
`namevec`	Character vector, giving the variable names with respect to which derivatives will be computed
`hessian`	Logical whether or not the second derivatives should be computed and returned

- Returns an unevaluated call for computing the `expr` along with a `gradient` attribute containing the gradient matrix and if `hessian=TRUE` a `hessian` attribute containing the Hessian array

Example - Symbolic Derivatives

```
# Formula argument
dx.expr <- deriv(~x^2, "x", hessian=TRUE)  # Unevaluated expression
dx.expr

x <- 1:5
dx.num <- eval(dx.expr)  # Evaluated expression
dx.num
attr(dx.num, "gradient") # Gradient Vector

# Expression argument
dx.dy.expr <- deriv(expression(x^2*y^4), c("x", "y"), hessian=TRUE)
dx.dy.expr

x <- 1:5; y <- 1:5
dx.dy.num <- eval(dx.dy.expr)
dx.dy.num
attr(dx.dy.num, "hessian") # For matrix "x", the columns are
                           # d2/dx2 and d2/dxdy
```

Root Finding

- The function `uniroot()` searches an interval for the root of a function

```
uniroot(f, interval, ...)
```

`f`	Function to get the root of
`interval`	Vector giving the interval `c(lower, upper)` to be searched
`...`	Additional arguments to be passed to `f`

- *Only returns one root*, does not return multiple roots, if multiple roots exist

Example - Root Finding

```
# Function we want to find the roots of
f <- function(x) {-x^4-.5*x^3+9*x^2-x-5}

# Plot the function we want to find the roots of
curve(f(x), -3, 3, lwd=2, main=expression(f(x)==-x^4-.5*x^3+9*x^2-x-5))
abline(h=0, lty=2, lwd=1)
points(c(2.563, .866, -.697), c(0,0,0), pch=19, cex=1.5, col="blue")

# Find the root between 0 and 2
uniroot(f, c(0, 2))

# Only returns one root, even if there are multiple roots in the interval
uniroot(f, c(-3, 3))

# The f values at the end points need to be of opposite sign
uniroot(f, c(1, 2))
```

Example - Simulating Survival Data

- Suppose we want to simulate $n = 100$ observations. The event times T follow a Weibull distribution with shape parameter $a = 2$ and scale parameter $b = 1$ and the censoring times C are distributed uniformly from 0 to τ. Then the observed time is, $X = \min(T, C)$. Assume T and C are independent.
- The value of τ is chosen to achieve the desired censoring rate. For this example we want 25% of the observations to be censored. Thus, $P(C < T) = 0.25$ and we need to choose τ such that,

$$\begin{aligned} P(C < T) &= \int_0^\tau \int_c^\infty f_T(t) f_C(c)\, dt\, dc \\ &= \frac{1}{\tau} \int_0^\tau \int_c^\infty f_T(t)\, dt\, dc \\ &= \frac{1}{\tau} \int_0^\tau S_T(c)\, dc \\ &= 0.25 \end{aligned}$$

Example - Simulating Survival Data

- Find τ by using `integrate()` and `uniroot()` to solve,

$$\frac{1}{\tau}\int_0^{\tau} S_T(c)\, dt\, dc - 0.25 = 0$$

```
# Calculate tau for a uniform(0, tau) censoring distribution in order to get the
# desired censoring rate (pctCensor) when the event times are Weibull(a, b)
uniformCensorUpperBound <- function(pctCensor, a, b) {
f <- function(tau) {
           S <- function(t) {1-pweibull(t, shape=a, scale=b)}
          (integrate(S, 0, tau)$value)/tau - pctCensor
      }
return(uniroot(f, c(0.001, 100))$root)
}

(tau <- uniformCensorUpperBound(.25, 2, 1))

# Verify
T <- rweibull(100, shape=2, scale=1)          # Event times
C <- runif(100, 0, tau)                       # Censor times
index <- apply(cbind(C, T), 1, which.min)-1   # Event indicator
table(index)/100
```

Optimization

- R has several functions for optimization,

`optimize()`	One dimensional optimization, no gradient or Hessian
`optim()`	General purpose optimization, five possible methods, gradient optional
`constrOptim()`	Minimization of a function subject to linear inequality constraints, gradient optional
`nlm()`	Non-linear minimization, can optionally include the gradient and hessian of the function as attributes of the objective function
`nlminb()`	Minimization using PORT routines, can optionally include the gradient and Hessian of the objective function as additional arguments

- These functions use different algorithms and accept different arguments, no one function is superior to the others. Which function you use depends on your particular problem; use `optimize()` for one-dimensional problems.
- To turn a minimization problem into a maximization problem, multiply the objective function and gradient by -1
- There are also packages with additional optimization functions, http://cran.r-project.org/web/views/Optimization.html

One-Dimensional Optimization

```
optimize(f, interval, ..., maximum=FALSE)
```

`f`	Function to be optimized
`interval`	Vector giving the interval `c(lower, upper)` to be searched
`...`	Additional arguments to be passed to `f`
`maximum`	Logical, find maximum if `TRUE`

- Cannot specify the gradient to assist with the optimization

Example - optimize()

```
# Objective function
f <- function(x) {-x^4-.5*x^3+9*x^2-x-5}

# Plot the objective function
curve(f(x), -3, 3, lwd=2, main=expression(f(x)==-x^4-.5*x^3+9*x^2-x-5))
abline(h=0, lty=2, lwd=1)

# Find the minimum of f
(min.f <- optimize(f, c(-3, 3)))
points(min.f$minimum, min.f$objective, pch=19, cex=1.5, col="blue")

# Find the maximum of f
(max.f <- optimize(f, c(-3, 3), maximum=TRUE))
points(max.f$maximum, max.f$objective, pch=19, cex=1.5, col="green3")

# Becareful, may not get a global max
(max.f <- optimize(f, c(1, 3), maximum=TRUE))
points(max.f$maximum, max.f$objective, pch=19, cex=1.5, col="red")
```

Multi-Dimensional Optimization

```
optim(par, fn, gr=NULL, ..., method, control)
```

`par`	Initial values
`fn`	Function to be optimized, argument is a vector of parameters
`gr`	A function that returns the gradient, same argument as `fn`
`...`	Additional arguments passed to `fn`
`method`	Method to be used
`control`	List of control parameters (number of iterations, tolerance, etc.)

- By default the minimum is found, to find the maximum set the control parameter `fnscale` to -1, `control=list(fnscale=-1)`. This divides the objective function and gradient by -1.

Example - optim()

```
# Two-dimensional objective function, the argument needs to be a vector
f <- function(x) {
     x1 = x[1]
     x2 = x[2]
     z = 10*x1^2*x2 - 5*x1^2 - 4*x2^2-x1^4-2*x2^4
     return(z)
}

# Plot the objective function
x <- y <- seq(-4, 4, len=50)
z <- outer(x, y, FUN=function(x,y) apply(cbind(x,y), 1, f))
filled.contour(x, y, z,
   color.palette=colorRampPalette(c("blue4","blue3","white","green3","green4")))

# Find the maximum points, from the contour plot there appears to be two maximums
# convergence=0 means convergence
# convergence=1 means reached max number of iterations
(pt1 <- optim(c(-2, 2), f, control=list(fnscale=-1)))
(pt2 <- optim(c(2, 2), f, control=list(fnscale=-1)))
```

Example - optim()

```
# Plot maximum points on contour plot
filled.contour(x, y, z,
   color.palette=colorRampPalette(c("blue4","blue3","white","green3","green4")),
   plot.axes={
        axis(1); axis(2)
        points(pt1$par[1], pt1$par[2], pch=19, col="red", cex=1.5)
        text(pt1$par[1], pt1$par[2], round(pt1$value, 3), pos=1)
        points(pt2$par[1], pt2$par[2], pch=19, col="red", cex=1.5)
        text(pt2$par[1], pt2$par[2], round(pt2$value, 3), pos=1)
     })

# Find maximum points using gradient
g <- function(x) {
     x1 = x[1]
     x2 = x[2]
     dx1 <- 20*x1*x2 - 10*x1 - 4*x1^3
     dx2 <- 10*x1^2 - 8*x2 - 8*x2^3
     return(c(dx1, dx2))
}

# Not all methods use the gradient, default method "Nelder-Mead" does not
optim(c(-2, 2), fn=f, gr=g, method="BFGS", control=list(fnscale=-1))
```

Constrained Optimization

- For box constraints use the `"L-BFGS-B"` method in `optim()` and the arguments `lower` and `upper` to give bounds for the arguments
- For linear inequality constraints use `constrOptim()`

```
constrOptim(theta, f, grad, ui, ci, control, ...)
```

`theta`	Starting values, $p \times 1$ vector
`f`	Function to be optimized
`grad`	Function that returns the gradient
`ui`	Constraint matrix, $k \times p$
`ci`	Constraint vector, $k \times 1$ vector
`control`	List of control parameters
`...`	Additional arguments passed to `f`

- Feasible region is defined by `ui %*% theta - ci >= 0`

Example - Constrained Optimization

```
f <- function(x) {          # Objective function
     x1 = x[1]; x2 = x[2]
     10*x1^2*x2 - 5*x1^2 - 4*x2^2-x1^4-2*x2^4
}
g <- function(x) {          # Gradient
     x1 = x[1]; x2 = x[2]
     dx1 <- 20*x1*x2 - 10*x1 - 4*x1^3; dx2 <- 10*x1^2 - 8*x2 - 8*x2^3
     return(c(dx1, dx2))
}
# Box constraints, x1>0
optim(c(1,0), f, lower=c(0, -Inf), method="L-BFGS-B", control=list(fnscale=-1))

# Inequality constraints, x1>=0 and x1>=x2
constrOptim(c(3,1), f, g, ui=matrix(c(1,1,0,-1), nrow=2), ci=c(0,0),
              control=list(fnscale=-1))

# Plot the objective function, with inequality constraints
x <- y <- seq(-4, 4, len=50)
z <- outer(x, y, FUN=function(x,y) apply(cbind(x,y), 1, f))
filled.contour(x, y, z, xlab=expression(x[1]), ylab=expression(x[2]),
   color.palette=colorRampPalette(c("blue4","blue3","white","green3","green4")),
   plot.axes={axis(1); axis(2)
              abline(a=0, b=1, lwd=2)
              lines(c(0,0), c(-4,0), lwd=2)
   })
```

Adaptive Rejection Sampling

- Suppose we are interested in generating a random sample from a distribution with density f. Unfortunately, this distribution is very difficult to sample from. However, there is another density g that is easy to sample from where for some constant c, $f(x) \leq cg(x)$ for all x.
- Thus we can generate a sample X from f by using rejection sampling,
  1. Generate Y from g
  2. Generate a random uniform(0,1) number U
  3. If $U \leq f(Y)/(cg(Y))$ set X=Y
- It may be difficult to determine g, in this case use adaptive rejection sampling.
- Adaptive rejection sampling (ARS) is a method that performs rejection sampling where the envelope function g is constructed and refined during the sampling process.
- Method is only applicable to univariate probability density functions that are **log-concave**.

Adaptive Rejection Sampling

- The function `ars()` in the `ars` package performs ARS

```
ars(n=1, f, fprima, ...)
```

`n`	Sample size
`f`	Function that returns log(f(u,...)), where f(u) is the density we want to sample from
`fprima`	d/du log(f(u,...))
`...`	Additional arguments to be passed to `f`

Example - Adaptive Rejection Sampling

```
library(ars)

# Mixture of two normal distributions with equal weight
f <- function(x) {.5*dnorm(x) + .5*dnorm(x, mean=2)}

integrate(f, -Inf, Inf)   # Density integrates to 1
curve(f(x), -3, 5, lwd=2) # Density shape

# Log-density
f.log <- function(x) {log(f(x))}
curve(f.log, -3, 5, lwd=2)  # Log-concave down

# Derivative of the log-density
f.log.prime <- function(x) {(-x*.5*dnorm(x) + -(x-2)*.5*dnorm(x, mean=2))/f(x)}

# ARS sample
data <- ars(10000, f.log, f.log.prime)

# Verify
hist(data, freq=FALSE)
curve(f(x), lwd=2, col="blue", add=TRUE)
```