Numerical Integration

Suppose you want to integrate a real valued function f from a to b. The trapezoidal rule based on k strips estimates the integral to be the sum of the areas of k trapezoids:

$$I_k = \sum_{i=1}^{k} \frac{f(x_{i-1}) + f(x_i)}{2}(x_i - x_{i-1})$$

where the ith strip runs from x_{i-1} to x_i. Usually we take $x_i - x_{i-1}$ to be $(b-a)/k$ so that all the strips have the same width.

It turns out that

$$I_k = I_\infty + C/k^2 + o(1/k^2)$$

where

$$I_\infty = \int_a^b f\,.$$

So if I know I_k and I_{2k} I can try pretending the o term is actually 0 and solving the two equations

$$I_k = I_\infty + C/k^2$$

and

$$I_{2l} = I_\infty + C/(4k^2)$$

to get

$$I_\infty = (4I_{2k} - I_k)/3$$

This idea produces a better estimate of the integral. (It's actually Simpson's rule.)

If you knew I_k, I_{2k} and I_{4k} you could get two new estimates of the integral. These estimates have an error whose asymptotic behaviour (as $k \to \infty$) is known and you can use the trick above to improve the estimate. This process, followed to its logical limit is Romberg extrapolation.

Here are some S functions which implement this:

•**The trapezoidal rule**: *trap*

This function computes I_1, I_2, and so on to I_{2^n-1}. It permits f to have values which are arranged in an array (usually just a vector) and expects that the the syntax `f(x)` returns an array whose components are $\sum_x$ of the values of that component of f.

```
trap<-
  function(f, a = 0, b = 1, n = 10, prev.value, ...)
  {
  # This Function integrates f from a to b.  It expects that
  # if x is a vector of values between a and b then
  # f(x,...) returns the sum of values of f for those x's
  # It can actually handle vector valued functions f. In
  # that case f(x,...) must return a vector of values. The
  # ith of these values is to be \sum_x f_i(x)
  #
```

```
  # The function uses the trapezoidal rule and returns a vector
  # of values of the trapezoidal rules evaluated on 1, 2, 4, ..
  # to 2^{n-1} strips
  #
  # If you have previously calculated these quantities up to
  # 2^{k-1} strips you can put those values in prev.value
  # and then just continue the computation from there. This
  # can save some computing time
  #
if(missing(prev.value)) {
ends <- ((f(a, ...) + f(b, ...)) * (b - a))/2
ends <- as.array(ends)
d <- dim(ends)
l <- length(ends)
pv <- array(0, dim = c(l, n + 1))
pv[, 1] <- c(ends)
for(i in 1:n) {
h <- (b - a)/(2^i)
x <- h * (2 * (1:(2^(i - 1))) - 1)
odds <- c(f(x, ...))
pv[, i + 1] <- pv[, i]/2 + h * odds
}
}
else {
ndims <- length(dim(prev.value))
m <- dim(prev.value)[ndims] - 1
mm <- length(prev.value)/(m + 1)
if(m >= n)
return(prev.value)
pv <- array(0, dim = c(mm, n + 1))
d <- dim(prev.value)[.Uminus(ndims)]
pv[, 1:(m + 1)] <- matrix(c(prev.value), ncol = m + 1)
for(i in (m + 1):n) {
h <- (b - a)/(2^i)
x <- h * (2 * (1:(2^(i - 1))) - 1)
odds <- c(f(x, ...))
pv[, i + 1] <- pv[, i]/2 + h * odds
}
}
array(c(pv), dim = c(d, n + 1))
}
```

For instance, if we define

```
f<-function(x){
```

```
  c(sum(x),sum(x^2),sum(x^3))
}
```

then `trap(ftest,a=0,b=1)` produces the output

```
     [,1]   [,2]     [,3]      [,4]      [,5]      [,6]     [,7]      [,8]
[1,]  0.5 0.5000 0.500000 0.5000000 0.5000000 0.5000000 0.500000 0.5000000
[2,]  0.5 0.3750 0.343750 0.3359375 0.3339844 0.3334961 0.333374 0.3333435
[3,]  0.5 0.3125 0.265625 0.2539062 0.2509766 0.2502441 0.250061 0.2500153
          [,9]    [,10]     [,11]
[1,] 0.5000000 0.500000 0.5000000
[2,] 0.3333359 0.333334 0.3333335
[3,] 0.2500038 0.250001 0.2500002
```

Notice that the columns are getting closer and closer to 1/2, 1/3 and 1/4 which are the integrals of the components of f. The ith column is a trapezoidal rule for $\int_0^1 f$ based on 2^{i-1} strips.

•Error estimation for the trapezoidal rule: *extrap.trap*

This function takes the output of `trap` and produces the estimated integrals (the last column of the value of `trap` and the estimated errors which are just absolute differences between the last two columns.

```
extrap.trap<- function(table)
 {
 # This function takes a vector of trapezoidal rules and estimates
 # the absolute error (in a very simple minded way)
 # returning the integral and estimated errors in a list
 #
d <- dim(table)
l <- length(d)
n <- d[l]
tt <- array(c(table), dim = c(length(table)/n, n))
int <- array(tt[, n], dim = d[.Uminus(l)])
prev.int <- array(tt[, n - 1], dim = d[.Uminus(l)])
errs <- abs(int - prev.int)
list(integrals = int, errors = errs)
 }
> extrap.trap(trap(f,a=0,b=1))
$integrals:
 [1]       [2]       [3]
 0.5 0.3333335 0.2500002

$errors:
 [1]          [2]          [3]
   0 4.768372e-07 7.152557e-07
```

•**The trapezoidal rule again**: `integ.trap`

This function puts together `trap` and `extrap.trap` and chooses the number of strips to make the errors adequately small.

```
integ.trap <- function(f, a = 0, b = 1, tol = 10^(-7), nmin = 10)
 {
 # This function takes the vector of trapezoidal rule integrals
 # built on 1, 2, 4, ..., 2^{n-1} strips and estimates
 # the integral and the error in the integral. If the
 # estimated error in integrtating any component of f
 # is more than tol it makes n one larger and tries again.
 #
n <- nmin
table <- trap(f, a, b, n)
v <- extrap.trap(table)
while(any(v$err > tol)) {
n <- n + 1
table <- trap(f, a, b, n, table)
v <- extrap.trap(table)
}
v
 }
>integ.trap(f,a=0,b=1)
$integrals:
 [1]       [2]  [3]
 0.5 0.3333333 0.25

$errors:
 [1]          [2]          [3]
   0 2.980232e-08 4.470348e-08
```

•**Romberg extrapolation**: `extrap.romb`

This function takes the table of trapezoidal rules and carries out Romberg extrapolation

```
extrap.romb <- function(table)
 {
 # This function does Romberg extrapolation of a table of
 # trapezoidal rules.  It returns integral and estimated errors.
 #
d <- dim(table)
l <- length(d)
n <- d[l]
tt <- array(c(table), dim = c(length(table)/n, n))
rtab <- array(0, dim = c(dim(tt), n))
rtab[,  , 1] <- tt
```

```
for(m in 2:n) {
p <- 4^(m - 1)
rtab[, 1:(n - m + 1), m] <- (p * rtab[, 2:(n - m + 2), m - 1] -
rtab[, 1:(n - m + 1), m - 1])/(p - 1)
}
int <- array(rtab[, 1, n], dim = d[.Uminus(1)])
prev.int <- array(rtab[, 2, n - 1], dim = d[.Uminus(1)])
errs <- abs(int - prev.int)
list(integrals = int, errors = errs)
 }
> extrap.romb(trap(f,a=0,b=1))
$integrals:
 [1]       [2]  [3]
 0.5 0.3333333 0.25

$errors:
 [1] [2] [3]
   0   0   0
```

• **Romberg extrapolation**: `integ.romb`

This function does the same thing as integ.trap but for the Romberg integration method. It prints a bit more stuff.

```
integ.romb <- function(f, a = 0, b = 1, tol = 10^(-7), nmin = 5, ...)
 {
n <- nmin
    #
    # Compute a table of trapezoidal rules based on 1, 2, 4, ... 2^{n-1} strips
    #
table <- trap(f, a, b, n,  , ...)
    #
    # Extrapolate the table of trapezoidal rules via Romberg estrapolation
    #
v <- extrap.romb(table)
while(any(v$err > tol)) {
    #
    # If any estimated error exceeds tol increase n and try again
    #
n <- n + 1
table <- trap(f, a, b, n, table, ...)
v <- extrap.romb(table)
}
v$table <- table
v
}
```

```
> integ.romb(ftest,a=0,b=1)
$integrals:
 [1]       [2]  [3]
 0.5 0.3333333 0.25

$errors:
 [1] [2] [3]
   0   0   0

$table:
     [,1]   [,2]     [,3]      [,4]      [,5]      [,6]
[1,]  0.5 0.5000 0.500000 0.5000000 0.5000000 0.5000000
[2,]  0.5 0.3750 0.343750 0.3359375 0.3339844 0.3334961
[3,]  0.5 0.3125 0.265625 0.2539062 0.2509766 0.2502441
```

```
trap<- function(f, a = 0, b = 1, n = 10, prev.value, ...){
if(missing(prev.value)) {
ends <- ((f(a, ...) + f(b, ...)) * (b - a))/2
ends <- as.array(ends)
d <- dim(ends)
l <- length(ends)
pv <- array(0, dim = c(l, n + 1))
pv[, 1] <- c(ends)
for(i in 1:n) {
h <- (b - a)/(2^i)
x <- h * (2 * (1:(2^(i - 1))) - 1)
odds <- c(f(x, ...))
pv[, i + 1] <- pv[, i]/2 + h * odds
}
} else {
ndims <- length(dim(prev.value))
m <- dim(prev.value)[ndims] - 1
mm <- length(prev.value)/(m + 1)
if(m >= n)
return(prev.value)
pv <- array(0, dim = c(mm, n + 1))
d <- dim(prev.value)[.Uminus(ndims)]
pv[, 1:(m + 1)] <- matrix(c(prev.value), ncol = m + 1)
for(i in (m + 1):n) {
h <- (b - a)/(2^i)
x <- h * (2 * (1:(2^(i - 1))) - 1)
odds <- c(f(x, ...))
pv[, i + 1] <- pv[, i]/2 + h * odds
}
}
array(c(pv), dim = c(d, n + 1))
}
> f<-function(x){
  c(sum(x),sum(x^2),sum(x^3))
}
> trap(f,0,1)
     [,1]   [,2]     [,3]      [,4]      [,5]      [,6]     [,7]      [,8]
[1,]  0.5 0.5000 0.500000 0.5000000 0.5000000 0.5000000 0.500000 0.5000000
[2,]  0.5 0.3750 0.343750 0.3359375 0.3339844 0.3334961 0.333374 0.3333435
[3,]  0.5 0.3125 0.265625 0.2539062 0.2509766 0.2502441 0.250061 0.2500153
          [,9]    [,10]     [,11]
[1,] 0.5000000 0.500000 0.5000000
[2,] 0.3333359 0.333334 0.3333335
[3,] 0.2500038 0.250001 0.2500002
```

```
extrap.trap<- function(table)
 {
 # This function takes a vector of trapezoidal rules and estimates
 # the absolute error (in a very simple minded way)
 # returning the integral and estimated errors in a list
 #
d <- dim(table)
l <- length(d)
n <- d[l]
tt <- array(c(table), dim = c(length(table)/n, n))
int <- array(tt[, n], dim = d[.Uminus(l)])
prev.int <- array(tt[, n - 1], dim = d[.Uminus(l)])
errs <- abs(int - prev.int)
list(integrals = int, errors = errs)
 }
> extrap.trap(trap(f,a=0,b=1))
$integrals:
 [1]       [2]       [3]
 0.5 0.3333335 0.2500002

$errors:
 [1]          [2]          [3]
   0 4.768372e-07 7.152557e-07
```

```
integ.trap <- function(f, a = 0, b = 1, tol = 10^(-7), nmin = 10)
 {
 # This function takes the vector of trapezoidal rule integrals
 # built on 1, 2, 4, ..., 2^{n-1} strips and estimates
 # the integral and the error in the integral. If the
 # estimated error in integrtating any component of f
 # is more than tol it makes n one larger and tries again.
 #
n <- nmin
table <- trap(f, a, b, n)
v <- extrap.trap(table)
while(any(v$err > tol)) {
n <- n + 1
table <- trap(f, a, b, n, table)
v <- extrap.trap(table)
}
v
 }
>integ.trap(f,a=0,b=1)
$integrals:
 [1]       [2]  [3]
 0.5 0.3333333 0.25

$errors:
 [1]          [2]          [3]
   0 2.980232e-08 4.470348e-08
```

```
extrap.romb <- function(table)
 {
 # This function does Romberg extrapolation of a table of
 # trapezoidal rules.   It returns integral and estimated errors.
 #
d <- dim(table)
l <- length(d)
n <- d[l]
tt <- array(c(table), dim = c(length(table)/n, n))
rtab <- array(0, dim = c(dim(tt), n))
rtab[,  , 1] <- tt
for(m in 2:n) {
p <- 4^(m - 1)
rtab[, 1:(n - m + 1), m] <- (p * rtab[, 2:(n - m + 2), m - 1] -
rtab[, 1:(n - m + 1), m - 1])/(p - 1)
}
int <- array(rtab[, 1, n], dim = d[.Uminus(l)])
prev.int <- array(rtab[, 2, n - 1], dim = d[.Uminus(l)])
errs <- abs(int - prev.int)
list(integrals = int, errors = errs)
 }
> extrap.romb(trap(f,a=0,b=1))
$integrals:
 [1]       [2]  [3]
 0.5 0.3333333 0.25

$errors:
 [1] [2] [3]
   0   0   0
```

```
integ.romb <- function(f, a = 0, b = 1, tol = 10^(-7), nmin = 5, ...)
 {
n <- nmin
    #
    # Compute a table of trapezoidal rules based on 1, 2, 4, ... 2^{n-1} strips
    #
table <- trap(f, a, b, n,  , ...)
    #
    # Extrapolate the table of trapezoidal rules via Romberg estrapolation
    #
v <- extrap.romb(table)
while(any(v$err > tol)) {
    #
    # If any estimated error exceeds tol increase n and try again
    #
n <- n + 1
table <- trap(f, a, b, n, table, ...)
v <- extrap.romb(table)
}
v$table <- table
v
}
> integ.romb(ftest,a=0,b=1)
$integrals:
 [1]       [2]  [3]
 0.5 0.3333333 0.25

$errors:
 [1] [2] [3]
   0   0   0

$table:
     [,1]   [,2]     [,3]      [,4]      [,5]      [,6]
[1,]  0.5 0.5000 0.500000 0.5000000 0.5000000 0.5000000
[2,]  0.5 0.3750 0.343750 0.3359375 0.3339844 0.3334961
[3,]  0.5 0.3125 0.265625 0.2539062 0.2509766 0.2502441
```