**Numerical integration**

The need to integrate a function in statistics is very common. Unfortunately, even for simple situations, like for the normal density function, integration cannot be done by hand. Rather, numerical integration techniques are needed to develop a good approximation in these situations. The purpose of this section is to examine the basics of numerical integration.

integrate() function

The stats package contains the integrate() function to perform integration. Inside its code, you will notice a call to a C function which does the actual calculations. This will make the function faster than if it was solely coded in R.

The syntax for the function is

integrate(f, lower, upper, ..., subdivisions = 100L,

 rel.tol = .Machine$double.eps^0.25, stop.on.error = TRUE)

The key argument is f which corresponds to the function being integrated. Note that this function needs to accept vector input (e.g., c(1,2,3,4)). The lower and upper arguments correspond to the lower and upper limits, respectively. Use -Inf or Inf when wanting to include infinity for a limit. We will discuss the subdivisions argument later.

Example: Beta distribution (beta.R)

Below are some simple examples involving probabilities from a beta distribution:

> a <- 0.5

> b <- 2

> #Use as ad-hoc way to get an upper limit for the plot

> ylim.max <- max(dbeta(x = 0.01, shape1 = a, shape2 = b),

 dbeta(x = 0.99, shape1 = a, shape2 = b))

> par(xaxs = "i") #Remove extra 4% around x-axis limits

> curve(expr = dbeta(x, shape1 = a, shape2 = b), xlim =

 c(0,1), ylab = "f(x)", xlab = "x", ylim = c(0,

 ylim.max), col = "red", main = substitute(paste("Beta

 distribution with ", alpha == a, " and ", beta == b),

 list(a = a, b = b)))

> abline(h = 0)

> #Density to integrate – because dbeta() already exists,

 do not really need to program in like this. However,

 this will be useful to see for later.

> beta.density <- function(x, a, b) {

 #Could program in density directly, but may be more

 likely to lead to a numerical error

 #1/beta(a = a, b = b) \* x^(a-1) \* (1-x)^(b-1)

 dbeta(x = x, shape1 = a, shape2 = b)

 }

> #Integrates to 1 over entire region

> integrate(f = beta.density, lower = 0, upper = 1, a = a,

 b = b)

1 with absolute error < 7.1e-06

> #Integrate over a specific region

> lower.bound <- 0.1

> upper.bound <- 0.5

> integrate(f = beta.density, lower = lower.bound, upper =

 upper.bound, a = a, b = b)

0.4253532 with absolute error < 7.4e-11

> #Check

> pbeta(q = upper.bound, shape1 = a, shape2 = b) - pbeta(q

 = lower.bound, shape1 = a, shape2 = b)

[1] 0.4253532

> #Shaded area on plot - notice duplicate x values at

> # the beginning and end of vector in polygon().

> # This is needed to correspond to the lower y limits

> x.values <- seq(from = lower.bound, to = upper.bound, by

 = 0.001)

> polygon(x = c(lower.bound, x.values, upper.bound), border

 = NA, y = c(0, dbeta(x.values, shape1 = a, shape2 =

 b),0), col = "blue")



> #Show how to save results

> ans <- integrate(f = beta.density, lower = lower.bound,

 upper = upper.bound, a = a, b = b)

> ans

0.4253532 with absolute error < 7.4e-11

> names(ans)

[1] "value" "abs.error" "subdivisions" "message"

 "call"

> ans$value

[1] 0.4253532

> ans$subdivisions

[1] 1

> ans$message

[1] "OK"

Therefore, .

Moments can be easily estimated by changing the function to be integrated:

> #E(X)

> beta.density.mu <- function(x, a, b) {

 x \* dbeta(x = x, shape1 = a, shape2 = b)

 }

> mu.x <- integrate(f = beta.density.mu, lower = 0, upper =

 1, a = a, b = b)

> mu.x

0.2000001 with absolute error < 6.1e-05

> a/(a+b)

[1] 0.2

> #Var(X)

> beta.density.var <- function(x, a, b, mu) {

 (x - mu)^2 \* dbeta(x = x, shape1 = a, shape2 = b)

 }

> var.x <- integrate(f = beta.density.var, lower = 0, upper

 = 1, a = a, b = b, mu = mu.x$value)

> var.x

0.04571429 with absolute error < 3.1e-06

> a\*b/ ((a+b)^2 \* (a+b+1))

[1] 0.04571429

Trapezoidal rule

The actual numerical methods to perform integration are based on partitioning the interval of integration into subintervals. Within these smaller intervals, approximations to the function of interest take place to make it easier to find the area of a region between the function and the x-axis.

Example: Beta distribution (beta.R)

This example shows how the integral



can be approximated by the trapezoidal rule. This method simply sums the areas of trapezoids. Below is the plot we saw previously with two trapezoids:



The tops of the trapezoids provide straight-line approximations to the actual beta distribution. The sum of the areas for these trapezoids gives the approximation to the actual area. Below is how the trapz() function in R can be used to calculate this:

> library(pracma)

> x.values2 <- seq(from = lower.bound, to = upper.bound, by

 = 0.2)

> trapz(x = x.values2, y = dbeta(x = x.values2, shape1 = a,

 shape2 = b))

[1] 0.458189645777165

> #Area of two trapezoids - 0.5 \* width \* (height1 +

 height2)

> 0.5 \* 0.2 \* (dbeta(x = 0.1, shape1 = a, shape2 = b) +

 dbeta(x = 0.3, shape1 = a, shape2 = b)) +

 0.5 \* 0.2 \* (dbeta(x = 0.3, shape1 = a, shape2 = b) +

 dbeta(x = 0.5, shape1 = a, shape2 = b))

[1] 0.458189645777165

> #Rewrite above in a simplified form

> 0.5 \* 0.2 \* dbeta(x = 0.1, shape1 = a, shape2 = b) +

 1 \* 0.2 \* dbeta(x = 0.3, shape1 = a, shape2 = b) +

 0.5 \* 0.2 \* dbeta(x = 0.5, shape1 = a, shape2 = b)

[1] 0.458189645777165

> #Alternative ways to write this that will be helpful

 shortly: SUM(w\_i \* f(x\_i))

> w1 <- 0.5 \* 0.2

> w2 <- 1 \* 0.2

> w3 <- 0.5 \* 0.2

> sum(w1\*dbeta(x = 0.1, shape1 = a, shape2 = b) +

 w2\*dbeta(x = 0.3, shape1 = a, shape2 = b) +

 w3\*dbeta(x = 0.5, shape1 = a, shape2 = b))

[1] 0.458189645777165

Thus, . Compare this result to what we obtained by integrate() which uses a more accurate numerical integration method.

Notice the last segment of code writes the integral approximation as  for n = 3.

Gaussian quadrature

Improvements can be made to the trapezoid rule. In particular, unequally spaced xi (points that f(⋅) is evaluated at) can be taken. This is the motivation behind using Gaussian quadrature methods and results in a weighted sum:



These weights, wi, and evaluation points for f(⋅), xi, are chosen so that the integral approximation will be as accurate as possible. The evaluation points are sometimes referred to as “nodes”.

The term “quadrature,” in general, simply means integrating. I see the term used most though in the context of numerical integration.

Example: Simple Gaussian quadrature

Suppose we would like to approximate



using n = 2 and we would like this approximation to be EXACT up to polynomials of order 3. We can then write



Solving the four equations for the four unknowns leads to w1 = 1, w2 = 1, x1 = , x2 = -x1.

On your own, calculate  using regular integration and the Gaussian quadrature to see that the same answer will result.

The restriction of a = -1 and b = 1 is not important. For other finite values of a and b, a simple “u substitution” of



can be used.

Gauss-Legendre quadrature

One way to approximate f(⋅) is to select xi = 1, …, n points to evaluate f(⋅) at and then connect the (xi, f(xi)) and (xi+1, f(xi+1)) pairs with straight lines. We could represent these straight lines by the notation Li(x). This is known as linear interpolation.

Higher degree polynomials can be used instead of the straight lines. In particular, Lagrange polynomials are of the following form:



Try a simple case with n = 3 to better understand the above expression if needed.

The function f(x) is approximately

 (1)

If we integrate from –1 to 1, we obtain

 (2)

Thus, we have the form of  that we saw earlier, which leads to the weights being



The choice of xi, for i = 1, …, n, is done to make the integral approximation as accurate as possible. For example, a remainder (error) term, say Rn(x), could be added to Equation (1) that leads to an equality:



for polynomial functions of order 2n+1. To make this remainder term be equal to 0, we can work with Legendre polynomials. Details are in Hultquist (1988).

Example: Beta distribution (beta.R)

Using n = 3, leads to

> library(pracma)

> save.xw <- gaussLegendre(n = 3, a = lower.bound, b =

 upper.bound)

> save.xw$x

[1] 0.145080666151704 0.300000000000000 0.454919333848297

> save.xw$w

[1] 0.111111111111111 0.177777777777777 0.111111111111111

> sum(save.xw$w \* dbeta(x = save.xw$x, shape1 = a, shape2 =

 b))

[1] 0.424790648740661

Thus,



where w1 = 0.1111, w2 = 0.1778, w3 = 0.1111, x1 = 0.1451, x2 = 0.3000, and x3 = 0.4549.

Using n = 7, leads to

> save.xw <- gaussLegendre(n = 7, a = lower.bound, b =

 upper.bound)

> save.xw$x

[1] 0.110178417531449 0.151693762880122 0.218830969724521

[4] 0.300000000000000 0.381169030275480 0.448306237119879

[7] 0.489821582468552

> save.xw$w

[1] 0.0258969932337736 0.0559410782978561 0.07636601010102

[4] 0.0835918367346939 0.0763660101010237 0.055941078297856

[7] 0.0258969932337739

> sum(save.xw$w \* dbeta(x = save.xw$x, shape1 = a, shape2 =

 b))

[1] 0.425353041658839

Gauss-Legendre quadrature is defined for the limits of integration of -1 to 1. Obviously, we used different limits here, and the function returned appropriate values. The actual weights for the entire -1 to 1 range are

> library(gaussquad)

> #x's and w's are not same as above because it assumes a

 range of -1 to 1

> leg.rule <- legendre.quadrature.rules(n = 3)

> leg.rule #Only look at [[3]]

[[1]]

 x w

1 0 2

[[2]]

 x w

1 0.577350269189626 1

2 -0.577350269189626 1

[[3]]

 x w

1 7.74596669241483e-01 0.555555555555556

2 7.77156117237610e-16 0.888888888888887

3 -7.74596669241482e-01 0.555555555555557

> #It appears that legendre.quadrature() transforms f(x) to

 correspond to the -1 to 1 limits.

> legendre.quadrature(functn = beta.density2, rule =

 leg.rule[[3]], lower = lower.bound, upper =

 upper.bound, alpha = a, beta = b)

[1] 0.424790648740661

Notice that

(2\*save.xw$x - (lower.bound + upper.bound))/(upper.bound

 - lower.bound)

[1] -7.74596669241482e-01 8.32667268468867e-16

[3] 7.74596669241484e-01

which shows how the x-values on the -1 to 1 interval and the x-values on the 0.1 to 0.5 interval are related (see the u-substitution formula given earlier). Also, notice that 0.1 to 0.5 is one-fifth the length of -1 to 1. So,

> save.xw$w \* 2/(upper.bound - lower.bound)

[1] 0.555555555555557 0.888888888888887 0.555555555555556

Other quadrature procedures

There are other quadrature procedures, and they each have their own settings where they work the best. Below is table from Bloomfield (2014, p. 138):



Unfortunately, Bloomfield does not define W(x) (it is not the wi that he uses in the book)! This W(x) represents part of the function being integrated. For example, an integral can be written as



Thus, if you have a normal density function, which will have some form of  over (-∞,∞), Gauss-Hermite quadrature will work well. Because Bayesian and GLMM methods often set a parameter or random effect to have a normal distribution, this type of quadrature can be quite accurate.

The integrate() function says the following about its use of quadrature:

For a finite interval, globally adaptive interval subdivision is used in connection with extrapolation by Wynn's Epsilon algorithm, with the basic step being Gauss–Kronrod quadrature.

The Gauss–Kronrod quadrature procedure is said to be “adaptive” in the sense that additional sets of x values are taken to estimate the error of the numerical integration.

I typically see “adaptive” to mean that a larger number of x values can be taken within certain intervals to obtain better estimates when needed. Givens and Hoeting (2012) briefly mention adaptive quadrature in Section 5.4.4 as does Bloomfield (2014) in Section 6.2. Based on my experience with using integrate(), I believe this is also occurring in it. The function allows for the recalculation of integral approximations until the error (as described in the previous paragraph) is less than the pre-stated threshold. Unfortunately, the function is not as well documented as maybe it should be, and the function calls a C program which I cannot read well ☹.

Example: Gauss-Hermite (GH.R)

There will be a few parts to this example. First, below is how I duplicated the example on p. 270 of McCulloch and Searle (2001). The purpose was to evaluate:



If we set g(x) = 1 + x2 and W(x) = , we have the form needed for Gauss-Hermite quadrature. Below is how we can perform the calculations in R:

> library(pracma)

> save.GH <- gaussHermite(n = 3)

> save.GH$x

[1] -1.224745e+00 8.881784e-16 1.224745e+00

> save.GH$w

[1] 0.295409 1.181636 0.295409

> x <- save.GH$x

> w <- save.GH$w

> sum((1 + x^2) \* w)

[1] 2.658681

> fx1 <- function(x) {

 (1 + x^2) \* exp(-x^2)

 }

> integrate(f = fx1, lower = -Inf, upper = Inf)

2.658681 with absolute error < 1.3e-06

The integral can be done without numerical integration. Below is how I evaluated it in Maple:

> fx:=(1+x^2)\*exp(-x^2);



> int(fx, x=-infinity..infinity);



> evalf(int(fx, x=-infinity..infinity));



Thus, a simple three-point Gauss-Hermite quadrature was able to obtain the correct value.

If we change the integral to



we see that 3-points was not enough to obtain the correct answer. However, 4-points were enough.

> sum((1 + x^6) \* w)

[1] 3.766464

> save.GH <- gaussHermite(n = 4)

> x <- save.GH$x

> w <- save.GH$w

> sum((1 + x^6) \* w)

[1] 5.095805

> fx2 <- function(x) {

 (1 + x^6) \* exp(-x^2)

 }

> integrate(f = fx2, lower = -Inf, upper = Inf)

5.095805 with absolute error < 0.00017

> fx:=(1+x^6)\*exp(-x^2);



> int(fx, x=-infinity..infinity);



> evalf(int(fx, x=-infinity ..

 infinity));



Next, let’s apply these same ideas to a normal distribution calculation. Suppose X ~ N(0, σ2), and we want to find E(X). This can be written as



To obtain the necessary form for Gauss-Hermite quadrature, let . Then  and . Our integral becomes



Suppose σ = 10. Using 2-point Gauss-Hermite quadrature, we obtain:

> save.GH <- gaussHermite(n = 2)

> u <- save.GH$x

> w <- save.GH$w

> sigma <- 10

> sum((sqrt(2) \* sigma \* u)/sqrt(pi) \* w)

[1] 0

> fx2 <- function(x, sigma = 10) {

 x \* 1/(sqrt(2\*pi) \* sigma) \* exp(-x^2 /(2\*sigma^2))

 }

> integrate(f = fx2, lower = -Inf, upper = Inf)

0 with absolute error < 0

> #u substitution

> fu2 <- function(u, sigma = 10) {

 (sqrt(2) \* sigma \* u)/sqrt(pi) \* exp(-u^2)

 }

> integrate(f = fu2, lower = -Inf, upper = Inf)

0 with absolute error < 0

Multiple integration

Numerical integration over multiple integrals is often referred to as cubature. To perform cubature, one potentially could do one integral first via methods described previously and then do the second integral. Based on the R functions presented so far, this may be difficult due to the type of mathematical function or the limits of integration.

There are a few R functions available for cubature, and one of these are described next.

Example: Multivariate normal (mult\_normal.R)

The adaptIntegrate() function of the cubature package performs numerical integration in multidimensions. This function is an R “wrapper” to a C program that performs the main calculations. Details on the numerical methods used are at <http://ab-initio.mit.edu/wiki/index.php/Cubature> if you are interested.

Suppose . Below is a plot of the density (see program for code):



Below is how I verify that the volume is 1:

> library(mvtnorm)

> library(cubature)

> mult.norm.density <- function(x, mu, sigma) {

 dmvnorm(x = x, mean = mu, sigma = sigma)

 }

> adaptIntegrate(f = mult.norm.density, lowerLimit = c(-4,

 -4), upperLimit = c(4, 4), mu = mu, sigma = sigma)

$integral

[1] 0.9998732

$error

[1] 9.806793e-06

$functionEvaluations

[1] 2907

$returnCode

[1] 0

> adaptIntegrate(f = mult.norm.density, lowerLimit = c(-10,

 -10), upperLimit = c(10, 10), mu = mu, sigma = sigma)

$integral

[1] 1.000002

$error

[1] 9.827034e-06

$functionEvaluations

[1] 4131

$returnCode

[1] 0

I was unable to get infinite limits to work, so this is why I chose appropriate non-infinite limits.

Below is how I calculate moments:

> mult.norm.density1 <- function(x, mu, sigma) {

 x[1]\*dmvnorm(x = x, mean = mu, sigma = sigma)

 }

> mult.norm.density2 <- function(x, mu, sigma) {

 x[2]\*dmvnorm(x = x, mean = mu, sigma = sigma)

 }

> mult.norm.density12 <- function(x, mu, sigma) {

 x[1]\*x[2]\*dmvnorm(x = x, mean = mu, sigma = sigma)

 }

> E.X1 <- adaptIntegrate(f = mult.norm.density1, lowerLimit

 = c(-4, -4), upperLimit = c(4, 4), mu = mu, sigma =

 sigma)

> E.X2 <- adaptIntegrate(f = mult.norm.density2, lowerLimit

 = c(-4, -4), upperLimit = c(4, 4), mu = mu, sigma =

 sigma)

> E.X1.X2 <- adaptIntegrate(f = mult.norm.density12,

 lowerLimit = c(-4, -4), upperLimit = c(4, 4), mu = mu,

 sigma = sigma)

> Cov.X1.X2 <- E.X1.X2$integral –

 E.X1$integral\*E.X2$integral

> data.frame(E.X1.X2$integral, E.X1$integral, E.X2$integral, Cov.X1.X2)

 E.X1.X2.integral E.X1.integral E.X2.integral Cov.X1.X2

1 0 0 0 0

Below is how I calculate a particular probability:

> adaptIntegrate(f = mult.norm.density, lowerLimit = c(0,

 0), upperLimit = c(4, 4), mu = mu, sigma = sigma)

$integral

[1] 0.2499683

$error

[1] 2.187174e-06

$functionEvaluations

[1] 731

$returnCode

[1] 0

> pmvnorm(lower = c(0, 0), upper = c(4, 4), mean = mu, sigma = sigma)

[1] 0.2499683

attr(,"error")

[1] 1e-15

attr(,"msg")

[1] "Normal Completion"

Bloomfield (2014) also mentions that the R2Cuba package can perform multiple integration. Additionally functions exist in the pracma package for two and three-dimensional integration. I have not tried any of these additional methods.

Monte Carlo Markov Chain (MCMC) and other simulation-based methods can provide easier ways to perform multiple integration for some problems.

Additional examples

Example: GLMM example (SimulateData\_FitModel.R)

Consider the simple GLMM model



where Yi ~ Binomial(ni,πi | β0,bi) and  for i = 1, …, n. The responses Yi are independent for i = 1, …, n. The bi are independent as well.

The ith contribution to the likelihood function is



The log likelihood function is



The maximum likelihood estimates of β0 and σ result from maximizing log(L). Using a large simulated data set, below is how I can find these estimates using the glmer() function of the lme4 package:

> n <- 1000 # Number of binomial observations

> beta0 <- 1

> sigma <- 2

> individual <- 1:n # Indicates observation number

> numb.trials <- 10 # n\_i - be careful with notation

> trials <- rep(x = numb.trials, times = n)

> # Simulate the response Y

> set.seed(8182)

> b <- rnorm(n = n, mean = 0, sd = sigma)

> pi.i <- plogis(beta0 + b)

> y <- rbinom(n = n, size = numb.trials, prob = pi.i)

> library(lme4)

> # Estimate model with 20-point quadrature

> mod.fit20 <- glmer(formula = y/trials ~ 1 +

 (1|individual), nAGQ = 20, weights = trials, family =

 binomial(link = "logit"))

> summary(mod.fit20)

Generalized linear mixed model fit by maximum likelihood ['glmerMod']

 Family: binomial ( logit )

Formula: y/trials ~ 1 + (1 | individual)

 AIC BIC logLik deviance

 2795.171 2804.987 -1395.586 2791.171

Random effects:

 Groups Name Variance Std.Dev.

 individual (Intercept) 3.966 1.992

Number of obs: 1000, groups: individual, 1000

Fixed effects:

 Estimate Std. Error z value Pr(>|z|)

(Intercept) 0.98927 0.06966 14.2 <2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> logLik(mod.fit20)

'log Lik.' -1395.586 (df=2)

The estimates are  and . As would be expected with a large sample size, these estimates are fairly close to their true values.

Below is how I estimate the parameters using integrate() and optim():

> my.func <- function(b, y, beta0, sigma, numb.trials) {

 pi.i <- plogis(beta0 + b)

 #Did not include the "n choose y" part

 pi.i^y \* (1 - pi.i)^(numb.trials - y) \* dnorm(x = b,

 mean = 0, sd = sigma)

 }

> log.Lik.int <- function(beta.sig, y, numb.trials) {

 beta0 <- beta.sig[1]

 sigma <- beta.sig[2]

 n <- length(y)

 L.i <- numeric(n)

 for(i in 1:n) {

 L.i[i] <- integrate(f = my.func, lower = -Inf, upper

 = Inf, y = y[i], beta0 = beta0, sigma = sigma,

 numb.trials = numb.trials)$value

 }

 sum(log(L.i)) #log(L)

 }

> log.Lik.int(beta.sig = c(beta0, sigma), y = y,

 numb.trials = numb.trials)

[1] -5216.237

> # Using the true values as the starting values;

 of course, this could not be done in practice

> save.opt.int <- optim(par = c(beta0, sigma), fn =

 log.Lik.int, method = "BFGS", y = y, numb.trials =

 numb.trials, control = list(fnscale = -1), hessian =

 TRUE)

> save.opt.int

$par

[1] 0.9892401 1.9915063

$value

[1] -5216.222

$counts

function gradient

 27 5

$convergence

[1] 0

$message

NULL

$hessian

 [,1] [,2]

[1,] -200.8032 37.4225

[2,] 37.4225 -205.7855

> cov.mat <- -solve(save.opt.int$hessian)

> sqrt(cov.mat[1,1]) # Estimate of Var(hat(beta)\_0)^0.5

[1] 0.07179621

> sqrt(cov.mat[2,2]) # Estimate of Var(hat(sigma))^0.5

[1] 0.07092175

The estimates are  and , which are very close to those found using glmer()! The standard error for  is similar for the two fitting methods as well. The log likelihood function evaluated at the parameter estimates is not the same for the two methods. What I expect is happening is that there are some constants not included in the log likelihoods for both R functions.

In order to use Gauss-Hermite quadrature directly, we need to re-write the integral



that is in the likelihood function. Similar to earlier, let. Then  and . The integral becomes



where



Below is my code:

> h.u <- function(u, y, beta0, sigma, numb.trials) {

 pi.tilde.i <- plogis(beta0 + u\*sigma\*sqrt(2))

 #Did not include the "n choose y" part

 1/sqrt(pi) \* pi.tilde.i^y \* (1 –

 pi.tilde.i)^(numb.trials-y)

 }

> library(pracma)

> save.GH <- gaussHermite(n = 20)

> u <- save.GH$x

> w <- save.GH$w

> log.Lik.GH <- function(beta.sig, y, u, w, numb.trials) {

 beta0 <- beta.sig[1]

 sigma <- beta.sig[2]

 n <- length(y)

 L.i <- numeric(n)

 for(i in 1:n) {

 L.i[i] <- sum(h.u(u = u, y = y[i], beta0 = beta0,

 sigma = sigma, numb.trials = numb.trials) \* w)

 }

 sum(log(L.i)) #log(L)

 }

> #Using the true values as the starting values; of course,

 this could not be done in practice

> save.opt.GH <- optim(par = c(beta0, sigma), fn =

 log.Lik.GH, method = "BFGS", y = y, u = u, w = w,

 numb.trials = numb.trials, control = list(fnscale =

 -1), hessian = TRUE)

> save.opt.GH

$par

[1] 1.012163 1.978129

$value

[1] -5216.547

$counts

function gradient

 26 5

$convergence

[1] 0

$message

NULL

$hessian

 [,1] [,2]

[1,] -201.63972 58.09751

[2,] 58.09751 -233.47117

> sqrt(cov.mat[1,1]) # Estimate of Var(hat(beta)\_0)^0.5

[1] 0.07309157

> sqrt(cov.mat[2,2]) # Estimate of Var(hat(sigma))^0.5

[1] 0.06792641

We obtain similar answers to those found earlier. Below is a contour plot for the log likelihood function:

> beta0.seq <- seq(from = -1, to = 3, by = 0.1)

> sigma.seq <- seq(from = 0.1, to = 4, by = 0.1)

> save.log.Lik <- matrix(data = NA, nrow =

 length(beta0.seq), ncol = length(sigma.seq))

> i <- 1

> j <- 1

> for (beta0 in beta0.seq) {

 for (sigma in sigma.seq) {

 save.log.Lik[i,j] <- log.Lik.GH(beta.sig = c(beta0,

 sigma), y = y, u = u, w = w, numb.trials =

 numb.trials)

 j <- j + 1

 }

 j <- 1

 i <- i + 1

 }

> contour(x = beta0.seq, y = sigma.seq, z = save.log.Lik,

 xlab = expression(beta[0]), ylab = expression(sigma),

 levels = c(-7000, -6500, -6000, -5500, -5400, -5300, -

 5250, -5225))

> abline(v = save.opt.GH$par[1], lty = "dotted")

> abline(h = save.opt.GH$par[2], lty = "dotted")



It looks like we have a nice log likelihood function! The optim() function found the estimated correctly.

Let’s try fitting the model in SAS!? I exported the data from R into a text file named glmm\_data.txt. Below is my SAS code:

**data** set1;

 infile 'C:\Users\Chris\Desktop\glmm\_data.txt'

 firstobs=**2**;

 input y trials individual;

**run**;

**proc** **glimmix** data=set1 method=quad(qpoints = **20**);

 class individual;

 model y/trials = / solution dist=binomial link=logit;

 random int / subject = individual;

**run**;



<output edited>





The estimates are very similar to those obtained before. The log likelihood function evaluated at the parameter estimates is -2272.44, which is different than the values given in R.

Final notes on numerical integration

* The interval that one integrates over may contain infinity. Fortunately, integrate() deals with this issue relatively well, because you can just enter -Inf or Inf within the lower and upper arguments. Still, there may be problems with other functions, as we saw with adaptIntegrate(). In general, appropriate u-substitutions may be helpful to obtain new limits of integration. Specifically, Bloomfield (2014, p. 144) recommends u = 1/x2.
* The integrate() function has an argument named subdivisions. By default, it is set at 100 (100L in its help file means “100 is an integer”). This corresponds to the maximum number of xi to xi+1 breaks that the function can examine. When its estimated error is still above a set threshold after the maximum number of subdivisions is reached, integrate() will stop and say “maximum number of subdivisions reached.” Of course, one can increase the number of subdivisions in an attempt to solve the problem. If this does not work, other integration functions may be needed.
* Pay attention to the relative error given by integrate(). While this is not necessarily exact, one should be very cautious about using an integration result if the error is relatively large.
* Sections 6.2.2 and 6.2.5 of Bloomfield (2014) discuss how to handle functions that have discontinuities.
* Please see the numerical mathematics R task view at [http://cran.r-project.org/web/views/
NumericalMathematics.html](http://cran.r-project.org/web/views/NumericalMathematics.html) for more information on R’s integration capabilities.
* I am not aware of a function that performs symbolic integration in R. However, there are R interfaces available to software that do. Search for “symbolic” in the above TaskView for more information.