Assignment 6: Comparing Different Numerical Integration Methods With an Example

yifan

2025-03-26

Model Setup

Suppose that we have a Generalized Linear Mixed Model:

$$y_i|\mathbf{b} \sim Poi(\mu_i), log(\mu_i) = \mathbf{X_i}\beta + \mathbf{Z_i}\mathbf{b}, \mathbf{b} \sim N(\mathbf{0}, \Lambda^{-1}) \tag{1}$$

And that $y_i|\mathbf{b}$ are independent (actually, this condition isn't stated in the textbook, but I figure that we do need this for the joint distribution of $\mathbf{y}$ and $\mathbf{b}$).

Given this condition, we have:

$$f(\mathbf{y}, \mathbf{b}) = f(\mathbf{y}|\mathbf{b})f(\mathbf{b}) = \prod_i \frac{exp(\mathbf{x_i} + \mathbf{z_i}\mathbf{b})^{y_i}}{y_i!} exp(exp(\mathbf{x_i} + \mathbf{z_i}\mathbf{b})) \prod_j \sqrt{\frac{\lambda_j}{2\pi}} exp(-\frac{\lambda_j b_j^2}{2}) \tag{2}$$

Which means: (Note that in textbook, the term $-\frac{1}{2}\sum_j log(2\pi)$was omitted, which may be a minor error. This may cause some of my results to be different than the textbook.)

$$log(f(\mathbf{y}, \mathbf{b})) = \sum_i \{y_i(\mathbf{X_i}\beta + \mathbf{Z_i}\mathbf{b}) - exp(\mathbf{X_i} + \mathbf{Z_i}\mathbf{b}) - log(y_i!)\} - \frac{1}{2}\sum_j (\lambda_j b_j^2 - log(\lambda_j) + log(2\pi)) \tag{3}$$

Now, we want to compute $f_y(\mathbf{y}) = \int f(\mathbf{y}, \mathbf{b}) d\mathbf{b}$ using different numberical integration methods. To simplify the model, we assume that the fixed effects part of the model is $\beta_0 + \beta_1 x$, while the random effects part is given by two factor variables, z_1 and z_2, each with two levels.

Suppose further that the first two elements of $\mathbf{b}$ relate to z_1 and the remainder to z_2 and that $diag(\Lambda) = [\sigma_1^{-2}, \sigma_1^{-2}, \sigma_2^{-2}, \sigma_1^{-2}]$.

The R code given by the book simulates 50 data from such a model, where X is generated by $X \sim U(0, 1)$, $\sigma_1^2 = \sigma_2^2 = 1$, $\beta_0 = 0, \beta_1 = 3$:

```
set.seed(0);n <- 50
x <- runif(n)
z1 <- sample(c(0,1),n,replace=TRUE)
z2 <- sample(c(0,1),n,replace=TRUE)
b <- rnorm(4) ## simulated random effects
X <- model.matrix(~x)
Z <- cbind(z1,1-z1,z2,1-z2)
eta <- X%*%c(0,3) + Z%*%b
mu <- exp(eta)
y <- rpois(mu,mu)
```

The implementation of the log joint density is also given in the book (I added the constant term which is lacking from the book).

```
lf.yb0 <- function(y, b, theta, X, Z, lambda) {
  beta <- theta[1:ncol(X)]
  theta <- theta[-(1:ncol(X))]
  eta <- X %*% beta + Z %*% b
  mu <- exp(eta)
  lam <- lambda(theta, ncol(Z))
  d <- length(b)
  sum(y * eta - mu - lfactorial(y)) - sum(lam * b^2)/2 + sum(log(lam))/2 - (d/2) * log(2*pi)
}
```

Here, theta contains the fixed effect parameters, i.e. β and the parameters of Λ. lambda is the name of a function for turning the parameters of λ into its leading diagonal elements:

```
var.func <- function(theta,nb) c(theta[1],theta[1],theta[2],theta[2])
```

Now, we upgrade the function lf.yb0 to lf.yb, so that the function can accept a matrix $\mathbf{b}$, where each column is a different $\mathbf{b_i}$, as suggested in the textbook:

```
lf.yb <- function(y, b, theta, X, Z, lambda) {
  # Extract fixed effects parameters (beta)
  beta <- theta[1:ncol(X)]
  # Remaining parameters are for lambda function
  lambda_params <- theta[-(1:ncol(X))]

  # Compute linear predictor eta (supports matrix b)
  # Uses broadcasting: X%*%beta is vector, Z%*%b is matrix
  eta <- as.vector(X %*% beta) + Z %*% b  # Results in n × k matrix

  # Compute Poisson means
  mu <- exp(eta)

  # Get diagonal elements of Lambda matrix
  lam <- lambda(lambda_params, ncol(Z))

  # Compute Poisson part of log-likelihood
  # Repeats calculation for each column of b
  poisson_part <- colSums(y * eta - mu - lfactorial(y))

  # Compute Gaussian part of log-likelihood
  d <- nrow(b)  # Dimension of random effects
  gaussian_part <- -0.5 * colSums(lam * b^2) + 0.5 * sum(log(lam)) - (d/2) * log(2*pi)

  # Return joint log-density (vector of length k)
  poisson_part + gaussian_part
}
```

Quadrature Rules

First, we try brute force application of the midpoint rule. Though the integration domain is infinite, we can restrict the domain to $[-5, 5]^4$, as suggested by the textbook. However, we do not calculate $\int f(\mathbf{y}, \mathbf{b}) d\mathbf{b}$ directly. Instead, we calculate $log(\int f(\mathbf{y}, \mathbf{b}) d\mathbf{b})$, to avoid underflow to zero.

The following code is included in the textbook. Here, we use 10 mesh grids for each dimension, resulting in

10^4 points needed to calculate the integral. Here, if we denote the log density at each point as lf_i and the weight for each point as w_i, the idea is:

$$\begin{aligned}\log\left(\sum_i w_i \exp\left(lf_i \exp\left(lf_i - \max(lf_i)\right)\right)\right) + \max(lf_i) &= \log\left(\exp\left(-\max(lf_i) + \sum_i w_i \exp(lf_i)\right)\right) + \max(lf_i) \\ &= \log\left(\sum_i w_i \exp(lf_i)\right) \\ &\approx \log\left(\int f(\mathbf{y}, \mathbf{b}) d\mathbf{b}\right)\end{aligned} \tag{4}$$

aiming to avoid overflow when calculating $exp(lf_i)$ directly.

```
theta <- c(0,3,1,1)
nm <- 10;m.r <- 10
grid <- as.matrix(expand.grid(rep(list(seq(-5, 5, length.out=nm)), 4)))
bm <- list(X=grid, w=rep((10/nm)^4, nrow(grid)))
lf <- lf.yb(y,t(bm$X),theta,X,Z,var.func)
log(sum(bm$w * exp(lf-max(lf)))) + max(lf)
```

```
## [1] -117.7305
```

The slight difference with the textbook is due to the difference of the constant term in the log density function. To evaluate the performance of this brute method, we double nm:

```
nm <- 20;
grid <- as.matrix(expand.grid(rep(list(seq(-5, 5, length.out=nm)), 4)))
bm <- list(X=grid, w=rep((10/nm)^4, nrow(grid)))
lf <- lf.yb(y,t(bm$X),theta,X,Z,var.func)
log(sum(bm$w * exp(lf-max(lf)))) + max(lf)
```

```
## [1] -118.7903
```

We can see that the performance isn't so ideal, even with more than 10^4 function evaluations. In fact, it took me quite some time (about 1 second) to run the code when $nm = 20$.

Moreover, we can try the implementation of Gauss quadrature rules. We know the performance would be ideal if the function is smooth.

```
library(statmod)
```

```
## Warning:   'statmod' R 4.3.3
```

```
nm <- 10
gq <- gauss.quad(nm)
nodes_scaled <- gq$nodes * (m.r/2)
weights_scaled <- gq$weights * (m.r/2)

grid <- as.matrix(expand.grid(rep(list(nodes_scaled), 4)))
weights_grid <- apply(expand.grid(rep(list(weights_scaled), 4)), 1, prod)
lf <- lf.yb(y, t(grid), theta, X, Z, var.func)
log(sum(weights_grid * exp(lf - max(lf)))) + max(lf)
```

```
## [1] -136.95
```

Again, we double the value of nm:

```
nm <- 20
gq <- gauss.quad(nm)
```

```r
nodes_scaled <- gq$nodes * (m.r/2)
weights_scaled <- gq$weights * (m.r/2)

grid <- as.matrix(expand.grid(rep(list(nodes_scaled), 4)))
weights_grid <- apply(expand.grid(rep(list(weights_scaled), 4)), 1, prod)
lf <- lf.yb(y, t(grid), theta, X, Z, var.func)
log(sum(weights_grid * exp(lf - max(lf)))) + max(lf)
```

```
## [1] -125.3369
```

We can see that the performance of the Gauss quadrature rules is even worse than the performance of the midpoint rule, indicating that the integrand is in fact not very smooth.

Laplace Approximation

In class, we learned the Laplace approximation formula:

$$\int f(\mathbf{y},\mathbf{b})d\mathbf{b} \approx f(\mathbf{y},\hat{\mathbf{b}}_\mathbf{y})\frac{(2\pi)^{d/2}}{|\mathbf{H}|^{1/2}} \tag{5}$$

Here, $\hat{\mathbf{b}}_\mathbf{y}$ denotes the point that maximizes $f(\mathbf{y},\mathbf{b})$ given $\mathbf{y}$, and $\mathbf{H}$ is the Hessian of $-log(f)$ with respect to $\mathbf{b}$ evaluated at $\mathbf{y},\hat{\mathbf{b}}_\mathbf{y}$.

In order to find $\hat{\mathbf{b}}_\mathbf{y}$, we can use the Newton method. This calls for the need of the gradient and the Hessian of $f(\mathbf{y},\mathbf{b})$ with respect to $\mathbf{b}$. Here, we can calculate the gradient and the Hessian of $log(f(\mathbf{y},\mathbf{b}))$, since log-transformation is strictly monotonous. By simple calculation, we have:

$$\nabla_\mathbf{b} log f(\mathbf{y},\mathbf{b}) = \frac{\partial log(f(\mathbf{y},\mathbf{b}))}{\partial \mathbf{b}} = \sum_{i=1}^{50}(y_i \mathbf{Z_i}^T - \mathbf{Z_i}^T exp(\mathbf{X_i}\beta + \mathbf{Z_i b})) - \Lambda \mathbf{b} \tag{6}$$

To make things clearer, we denote the matrix $\mathbf{Z}$: $\begin{bmatrix}\mathbf{Z_1}\\ \vdots \\ \mathbf{Z_{50}}\end{bmatrix}$, and the matrix $\mathbf{X}$: $\begin{bmatrix}\mathbf{X_1}\\ \vdots \\ \mathbf{X_{50}}\end{bmatrix}$ and the gradient can be rewritten as:

$$\nabla_\mathbf{b} log f(\mathbf{y},\mathbf{b}) = \mathbf{Z}^T(\mathbf{y} - exp(\mathbf{X}\beta + \mathbf{Zb})) - \Lambda \mathbf{b} \tag{7}$$

In this way, the Hessian of $log(f(\mathbf{y},\mathbf{b}))$ can be written as: $\nabla^2_\mathbf{b} log(f(\mathbf{y},\mathbf{b})) = \frac{\partial^2 log(f(\mathbf{y},\mathbf{b}))}{\partial \mathbf{b}\partial \mathbf{b}^T} = -\mathbf{Z}^T\mathbf{D}\mathbf{Z} - \Lambda$ where $\mathbf{D} = diag(exp(\mathbf{X_1} + \mathbf{Z_1 b}), \cdots, exp(\mathbf{X_{50}} + \mathbf{Z_{50} b}))$. We implement the gradient and the Hessian using R code:

```r
glf.yb <- function(y, b, theta, X, Z, lambda) {
  # Extract fixed effects (beta) and parameters for Lambda
  beta <- theta[1:ncol(X)]
  lambda_params <- theta[-(1:ncol(X))]

  # Compute linear predictor eta and Poisson mean mu
  eta <- as.vector(X %*% beta) + Z %*% b  # X%*%beta is fixed, Z%*%b is random
  mu <- exp(eta)  # Poisson mean (exp(eta))

  # Get diagonal elements of precision matrix Lambda
  lam <- lambda(lambda_params, ncol(Z))  # e.g., c(sigma1^{-2}, sigma1^{-2}, sigma2^{-2}, sigma2^{-2})

  # Gradient calculation:
```

```
  # 1. Poisson part: Z^T (y - mu)
  # 2. Gaussian prior part: -Lambda * b
  gradient <- t(Z) %*% (y - mu) - lam * b

  return(gradient)
}
```

```
hlf.yb <- function(y, b, theta, X, Z, lambda) {
  # Extract fixed effects (beta) and parameters for Lambda
  beta <- theta[1:ncol(X)]
  lambda_params <- theta[-(1:ncol(X))]

  # Compute linear predictor eta and Poisson mean mu
  eta <- as.vector(X %*% beta) + Z %*% b
  mu <- exp(eta)  # exp(eta) for Poisson

  # Get diagonal elements of precision matrix Lambda
  lam <- lambda(lambda_params, ncol(Z))

  # Hessian calculation:
  # 1. Poisson part: -Z^T D Z (D = diag(mu))
  # 2. Gaussian prior part: -Lambda (already negative definite)
  D <- diag(c(mu))  # Diagonal matrix with mu as entries
  hessian <- - t(Z) %*% D %*% Z - diag(lam)

  return(hessian)
}
```

In this way, we can use the Newton method to find $\hat{\mathbf{b}}_\mathbf{y}$: first, we set $\mathbf{b} = [0, 0, 0, 0]^T$. Then, we conduct the iterative process:

$$\mathbf{b_{new}} = \mathbf{b_{old}} - \mathbf{H}^{-1}\mathbf{g} \tag{8}$$

Here, $\mathbf{g}, \mathbf{H}$ denotes the gradient and the Hessian of $log(f(\mathbf{y}, \mathbf{b}))$ with respect to $\mathbf{b}$. Note that instead of calculating the inverse of $\mathbf{H}$, we solve the system of linear equations: $\mathbf{H}\Delta\mathbf{b} = -\mathbf{g}$, which saves cumputation time.

```
b <- rep(0,4)
while(TRUE) {
b0 <- b
g <- glf.yb(y,b,theta,X,Z,var.func)
H <- hlf.yb(y,b,theta,X,Z,var.func)
b <- b - solve(H,g)
if (sum(abs(b-b0))<1e-10 * sum(abs(b))) break
}
```

And we obtain $\hat{\mathbf{b}}_\mathbf{y}$ and $\mathbf{H}$. It is important to note that $\mathbf{H}$ isn't the Hessian used in the Laplace approximation formula: the two differ by a minus sign. However, this wouldn't matter in this case, since the dimension of $\mathbf{b}$ is four, which means that $\mathbf{H}$ is a 4×4 matrix, and thus $|\mathbf{H}| = |-\mathbf{H}|$.

By simple calculation, we have $log(\int f(\mathbf{y}, \mathbf{b})d\mathbf{b}) \approx log(f(\mathbf{y}, \hat{\mathbf{b}}_\mathbf{y}) + \frac{d}{2}log(2\pi) - \frac{1}{2}log(|\mathbf{H}|)$:

```
H = -H
lf.yb(y,b,theta,X,Z,var.func) + 2 * log(2 * pi) - 0.5 * determinant(H,logarithm=TRUE)$modulus
```

```
## [1] -119.2717
## attr(,"logarithm")
## [1] TRUE
```

This is close to the truth, but we don't actually know how close.

Monte Carlo Method

Next, we will try some stochastic integration methods we have learned before. First is the Monte Carlo Method:

$$\int f(\mathbf{y},\mathbf{b})d\mathbf{b} \approx V\frac{1}{N}\sum_{i=1}^{N} f(\mathbf{y},\mathbf{b_i}) \tag{9}$$

Here, $\mathbf{b_i}$ follows high-dimensional uniform distribution, and V denotes the volume of the support of the uniform distribution. After performing log-transformation, we have:

$$log(\int f(\mathbf{y},\mathbf{b})d\mathbf{b}) \approx log(\sum_{i=1}^{N} f(\mathbf{y},\mathbf{b_i})) + log(V) - log(N) \tag{10}$$

Again, we restrict the domain to $[-5,5]^4$:

```
N <- 100000
bm <- matrix((runif(N * 4)-.5) * m.r,4,N)
lf <- lf.yb(y,bm,theta,X,Z,var.func)
log(sum(exp(lf-max(lf)))) + max(lf) - log(N) + log(m.r^4)
```

```
## [1] -119.8479
```

The result is close to what we obtained using Laplace approximation.

Laplace Important Sampling

Finally, we will try the Laplace important sampling method. Say we want to calculate $\int f(\mathbf{x})d\mathbf{x}$. The idea is that we need to find the proposal density for the importance sampling method, so naturally, we can use the normal density implied by Laplace approximation: $N(\hat{\mathbf{x}},\mathbf{H}^{-1})$, here $\hat{\mathbf{x}}$ denotes the value that maximizes the density f, and H is the Hessian of $-log(f)$. Plug this into the importance sampling method, and we obtain:

$$\int f(\mathbf{x})d\mathbf{x} \approx \frac{(2\pi)^{d/2}}{n\,|R|}\sum_{i=1}^{n} f(\hat{\mathbf{x}}+\mathbf{R}^{-1}z_i)exp(z_i^T z_i/2) \tag{11}$$

Here, we used a little trick: if $\mathbf{z}\sim N(\mathbf{0},I)$, then $\mathbf{X}=\hat{\mathbf{x}}+\mathbf{R}^{-1}\mathbf{z}\sim N(\hat{\mathbf{x}},\mathbf{H}^{-1})$, where $\mathbf{R}$ is a Choleski factor of $\mathbf{H}$, i.e. $R^TR=H$.

In our case, we want to calculate $\int f(\mathbf{y},\mathbf{b})d\mathbf{b}$, and the formula becomes:

$$\int f(\mathbf{y},\mathbf{b})d\mathbf{b} \approx \frac{(2\pi)^{d/2}}{n\,|R|}\sum_{i=1}^{n} f(\mathbf{y},\hat{\mathbf{b}}_\mathbf{y}+\mathbf{R}^{-1}z_i)exp(z_i^T z_i/2) \tag{12}$$

After a simple log-transformation, we obtain:

$$log(\int f(\mathbf{y},\mathbf{b})d\mathbf{b}) \approx log(\sum_{i=1}^{n} f(\mathbf{y},\hat{\mathbf{b}}_\mathbf{y}+\mathbf{R}^{-1}z_i)) + \frac{d}{2}log(2\pi) - log(n\,|R|) \tag{13}$$

The implementation using R is already given in the textbook (however, I do had to change the fourth line and put an "as.vector" function before b. This could be a minor error in the textbook.) Note that we applied the trick of subtracting max(lfz) to avoid overflow again.

```
N <- 10000
R <- chol(H)
z <- matrix(rnorm(N*4),4,N)
bm <- as.vector(b) + backsolve(R,z)
lf <- lf.yb(y,bm,theta,X,Z,var.func)
zz.2 <- colSums(z*z)/2
lfz <- lf +zz.2
log(sum(exp(lfz - max(lfz)))) + max(lfz) + 2*log(2 * pi) -sum(log(diag(R))) - log(N)
```

```
## [1] -119.2686
```

And we obtain the result. Note that this is even closer to the result we obtained by Laplace approximation.

Now, we double the value of N:

```
N <- 20000
z <- matrix(rnorm(N*4),4,N)
bm <- as.vector(b) + backsolve(R,z)
lf <- lf.yb(y,bm,theta,X,Z,var.func)
zz.2 <- colSums(z*z)/2
lfz <- lf +zz.2
log(sum(exp(lfz - max(lfz)))) + max(lfz) + 2*log(2 * pi) -sum(log(diag(R))) - log(N)
```

```
## [1] -119.2718
```

We can see that the result almost remains the same, indicating that it is very accurate. Thus, we have the confidence to believe that $log(\int f(\mathbf{y}, \mathbf{b})d\mathbf{b}) \approx -119.27$, indicating that $\int f(\mathbf{y}, \mathbf{b})d\mathbf{b} \approx exp(-119.27)$.

Conclusion

We can summarize our results into the table (Value denotes the approximation of $\int log(f(\mathbf{y}, \mathbf{b}))d\mathbf{b}$):

Method	Value	Bias
Truth	-119.27	0
Midpoint Rule	-117.73	1.54
Gauss Quadrature Rule	-136.95	-17.68
Laplace Approximation	-119.272	0.002
Monte Carlo Method	-119.347	0.077
Laplace Important Sampling	-119.269	0.001

It can be seen that in this example, the Laplace approximation method and the Laplace important sampling method generate the best results.