Aster Models and the Delta Method

Charles J. Geyer

December 7, 2025

Abstract

There are lots of ways to do the calculations involved in the delta method. Here we illustrate what is the easiest way to use the delta method to obtain standard errors for functions of parameters and random effects (if any) for models fit by R package `aster`.

1 License

2 R

- The version of R used to make this document is 4.5.2.
- The version of the `knitr` package used to make this document is 1.50.
- The version of the `aster` package used to make this document is 1.3.7.
- The version of the `numDeriv` package used to make this document is 2016.8.1.1.

```
> library(aster)
> library(numDeriv)
```

3 The Delta Method: Old Way and New Way

The first paper about aster models (Geyer, Wagenius, and Shaw, 2007, Section 3.3) already briefly mentions using the delta method along with the

method of R generic function `predict` that handles objects of class `aster` to obtain standard errors for nonlinear functions of parameters. And the technical report (Geyer, Wagenius, and Shaw, 2005, Appendix A) that provides supplementary material for that paper gives more details. That is the "old way" to apply the delta method to aster models.

Since that time, there have been a lot of changes to R package `aster` (Geyer, 2025). Aster models with random effects (Geyer, Ridley, Latta, Etterson, and Shaw, 2013) were added. So now we need to apply the delta method to obtain standard errors for estimates obtained from model fits of class `aster` and class `reaster`. But new tools have been developed. R package `aster` has gotten methods of R generic function `vcov` that handle R objects of class `aster` or class `reaster`. R package `numDeriv` has methods that calculate derivatives of nonlinear functions without users having to know (or explain) calculus.

So the "new way" to apply the delta method to aster models uses these new tools. This new way is exemplified in Geyer, Kulbaba, Sheth, Pain, Eckhart, and Shaw (2022) and Shaw, Geyer, Kulbaba, Sheth, Eckhart, and Pain (in preparation).

The delta method says that if a vector parameter estimate $\hat{\beta}$ has an approximate multivariate normal distribution with mean β (the true unknown vector parameter value) and variance-covariance matrix Σ and if g is a vector-to-vector function differentiable at β with derivative matrix (also called Jacobian matrix) $B = \nabla g(\beta)$, then the vector parameter estimate $g(\hat{\beta})$ has an approximate multivariate normal distribution with mean $g(\beta)$ and variance-covariance matrix $B \Sigma B^T$.

The "new way" gets Σ from R generic function `vcov`, gets the Jacobian matrix from R function `jacobian` in R package `numDeriv`, and does the matrix multiplication `B %*% Sigma %*% t(B)` explicitly in R.

For future reference (we use this in Section 5 below) we note that the delta method can be applied recursively. The composition of differentiable functions is differentiable, and the derivative of the composition is the matrix multiplication of the derivative (Jacobian matrices). If $g_3 = g_1 \circ g_2$ meaning

$$g_3(\beta) = g_1(g_2(\beta)), \qquad \text{for all } \beta$$

and

$$B_i = \nabla g_i(\beta)$$

then

$$B_3 = B_1 B_2$$

So the delta method says that the variance-covariance matrix of $g_3(\hat{\beta})$ is given by

$$(B_1 B_2)\Sigma(B_1 B_2)^T = B_1 B_2 \Sigma B_2^T B_1^T$$

which is just the delta method applied twice (to g_2 and then g_1).

4 Example I: No Random Effects

4.1 Introduction

We redo an example from a technical report (Geyer and Shaw, 2010) that is supplementary material for the paper Shaw and Geyer (2010). Because it was beyond what scientists had imagined could be done, that paper used simulated data to expand horizons. (Later papers did apply these methods to real data.)

```
> data(sim)
```

We fit a quadratic regression of fitness on phenotypic traits of an organism. The corresponding mean value parameters, considered as a function of phenotypic traits (predictor variables) is called the *fitness landscape*.

```
> aout <- aster(resp ~ varb + 0 + z1 + z2 + I(z1^2) + I(z1 * z2) + I(z2^2),
+     pred, fam, varb, id, root, data = redata)
> summary(aout)

Call:
aster.formula(formula = resp ~ varb + 0 + z1 + z2 + I(z1^2) +
    I(z1 * z2) + I(z2^2), pred = pred, fam = fam, varvar = varb,
    idvar = id, root = root, data = redata)

            Estimate Std. Error z value Pr(>|z|)
varbiflow1 -3.444251   0.180123 -19.122  < 2e-16 ***
varbiflow2 -3.064152   0.203311 -15.071  < 2e-16 ***
varbiflow3 -3.207467   0.218952 -14.649  < 2e-16 ***
varbiflow4 -3.284180   0.236597 -13.881  < 2e-16 ***
varbisurv1 -0.065167   0.160348  -0.406  0.68444
varbisurv2 -0.700847   0.225747  -3.105  0.00191 **
varbisurv3 -0.094013   0.275111  -0.342  0.73256
varbisurv4  1.217672   0.234288   5.197 2.02e-07 ***
varbnflow1 -7.264353   0.090581 -80.198  < 2e-16 ***
varbnflow2 -7.452760   0.102617 -72.627  < 2e-16 ***
```

```
varbnflow3 -7.227782   0.105711 -68.373  < 2e-16 ***
varbnflow4 -7.044131   0.107792 -65.349  < 2e-16 ***
varbngerm1 -2.264595   0.030308 -74.720  < 2e-16 ***
varbngerm2 -2.270312   0.033766 -67.237  < 2e-16 ***
varbngerm3 -2.325980   0.036102 -64.429  < 2e-16 ***
varbngerm4 -2.304824   0.036977 -62.332  < 2e-16 ***
varbnseed1  2.881224   0.009182 313.789  < 2e-16 ***
varbnseed2  2.895118   0.010258 282.241  < 2e-16 ***
varbnseed3  2.880964   0.010737 268.332  < 2e-16 ***
varbnseed4  2.864026   0.011117 257.619  < 2e-16 ***
z1          0.146950   0.013695  10.730  < 2e-16 ***
z2         -0.020598   0.009842  -2.093  0.03637 *
I(z1^2)    -0.027807   0.009508  -2.925  0.00345 **
I(z1 * z2)  0.023713   0.012352   1.920  0.05489 .
I(z2^2)    -0.017986   0.006536  -2.752  0.00593 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

4.2 Point Estimate

We want the maximum of the fitness landscape calculated by the function

```
> foo <- function(beta) {
+     A <- matrix(NaN, 2, 2)
+     A[1, 1] <- beta["I(z1^2)"]
+     A[2, 2] <- beta["I(z2^2)"]
+     A[1, 2] <- beta["I(z1 * z2)"] / 2
+     A[2, 1] <- beta["I(z1 * z2)"] / 2
+     b <- rep(NaN, 2)
+     b[1] <- beta["z1"]
+     b[2] <- beta["z2"]
+     solve(-2 * A, b)
+ }
```

Try it.

```
> foo(aout$coef)

[1] 3.335738 1.626314
```

This agrees with the “old way” shown in Geyer and Shaw (2010, Section 4).

4.3 Standard Errors

4.3.1 The New Way

The new way does

```
> Sigma <- vcov(aout)
> B <- jacobian(foo, aout$coef)
> V <- B %*% Sigma %*% t(B)
> V
```

```
          [,1]      [,2]
[1,] 1.1965982 0.8126606
[2,] 0.8126606 1.1215374
```

This disagrees with the calculation in Geyer and Shaw (2010, Section 4). Something is wrong. Either R function `jacobian` is wrong, or Geyer and Shaw botched their calculus (or their R implementation of that calculus). As we shall see, they did indeed botch the calculus.

4.3.2 The Old Way

This section can be skipped. This is the old way implemented correctly.

The old way extracts `aout$fisher` with the dollar sign operator rather than a helper function and then explicitly inverts this matrix

```
> all.equal(Sigma, solve(aout$fisher), check.attributes = FALSE)
```

```
[1] TRUE
```

That checks that the old way and the new way do the same thing for this part (asymptotic variance-covariance matrix coefficients vector).

Now we have to differentiate

$$c = -\tfrac{1}{2} A^{-1} b$$

with respect to any parameter β_k.

$$\frac{\partial c}{\partial \beta_k} = -\tfrac{1}{2} A^{-1} \frac{\partial b}{\partial \beta_k} + \tfrac{1}{2} A^{-1} \frac{\partial A}{\partial \beta_k} A^{-1} b$$

Do it.

```
> beta <- aout$coef
> A <- matrix(NaN, 2, 2)
> A[1, 1] <- beta["I(z1^2)"]
> A[2, 2] <- beta["I(z2^2)"]
> A[1, 2] <- beta["I(z1 * z2)"] / 2
> A[2, 1] <- beta["I(z1 * z2)"] / 2
> b <- rep(NaN, 2)
> b[1] <- beta["z1"]
> b[2] <- beta["z2"]
> jack <- matrix(0, nrow = nrow(B), ncol = ncol(B))
> # d b / d beta["z1"]
> i <- names(beta) == "z1"
> jack[ , i] <- solve(-2 * A, c(1, 0))
> # d b / d beta["z2"]
> i <- names(beta) == "z2"
> jack[ , i] <- solve(-2 * A, c(0, 1))
> # d A / d beta["I(z1^2)"]
> dA <- matrix(0, 2, 2)
> dA[1, 1] <- 1
> i <- names(beta) == "I(z1^2)"
> jack[ , i] <- (1 / 2) * solve(A) %*% dA %*% solve(A) %*% b
> # d A / d beta["I(z2^2)"]
> dA <- matrix(0, 2, 2)
> dA[2, 2] <- 1
> i <- names(beta) == "I(z2^2)"
> jack[ , i] <- (1 / 2) * solve(A) %*% dA %*% solve(A) %*% b
> # d A / d beta["I(z1 * z2)"]
> dA <- matrix(0, 2, 2)
> dA[1, 2] <- 1 / 2
> dA[2, 1] <- 1 / 2
> i <- names(beta) == "I(z1 * z2)"
> jack[ , i] <- (1 / 2) * solve(A) %*% dA %*% solve(A) %*% b
> all.equal(jack, B)

[1] TRUE
```

This check shows that R function `jacobian` in R package `numDeriv` is correct in its calculus.

4.3.3 Comment

The new way is a lot easier than the old way, which is so hard that it was botched by Geyer and Shaw (2010, Section 4).

4.3.4 Clean Up

Clear the R global environment, removing the trash from example I.

```
> rm(list = ls())
```

5 Example II: With Random Effects

5.1 Fit Aster Model with Random Effects

Fit a random effects aster model. Unfortunately, this takes more time than CRAN allows, so we precompute this result and save it (so CRAN does not notice). To see this actually work, remove the file `vignettes/rout.rda` from the aster package sources and rebuild this vignette.

```
> lout <- suppressWarnings(try(load("rout.rda"), silent = TRUE))
> if (inherits(lout, "try-error")) {
+     data(grey_cloud_2015)
+     modmat.sire <- model.matrix(~ 0 + fit:paternalID, redata)
+     modmat.dam <- model.matrix(~ 0 + fit:maternalID, redata)
+     modmat.siredam <- cbind(modmat.sire, modmat.dam)
+     rout.time <- system.time(
+         rout <- reaster(resp ~ fit + varb,
+             list(parental = ~ 0 + modmat.siredam, block = ~ 0 + fit:block),
+             pred, fam, varb, id, root, data = redata)
+     )
+     save(rout, rout.time, file = "rout.rda")
+ }
```

This invocation of R function `reaster` on this model for these data takes 24 minutes and 56.5 seconds (on one computer). Not really long. We've seen worse.

The results of such a fit are the following "estimates" in scare quotes: the vector `alpha` of fixed effects, the vector `b` of random effects, and the vector `nu` of variance components. The reason for the scare quotes is that α and ν are indeed parameters of the model and `rout$alpha` and `rout$nu` are the approximate maximum likelihood estimates (Geyer, et al., 2013) but, at

least according to the frequentist theory of statistics, the vector b of random effects is not a parameter of the model, and it makes no sense to estimate it, or, at least, it is very unclear what it means to "estimate" it (in scare quotes). What `rout$b` is is the conditional mode of the distribution of the random effects given the data and the parameters. This is, of course, a function of the parameters α and ν.

5.2 Asymptotic Variance-Covariance Matrix

So we can use the delta method to calculate a joint asymptotic variance-covariance matrix for the combined vector (α, b, ν), and this is what

```
> Sigma <- vcov(rout, standard.deviation = FALSE, re.too = TRUE)
```

Does for us.

5.3 Map Random Effect Estimates to Mean Values

The vector b of random effects is not scientifically interpretable. Hence we map it to the mean value parameter scale (response scale). In the process, we also correct the means for the artificial subsampling done in the experiment. For any more explanation of what is going on here, see Shaw, et al. (in preparation) and references cited therein.

The following R function is just copied from the supplementary material for Shaw, et al. (in preparation) (you are not expected to understand this)

```
> map.factory <- function(rout) {
+     stopifnot(inherits(rout, "reaster"))
+     aout <- rout$obj
+     stopifnot(inherits(aout, "aster"))
+     nnode <- ncol(aout$x)
+     nind <- nrow(aout$x)
+     fixed <- rout$fixed
+     random <- rout$random
+     if (nnode == 4) {
+         is.subsamp <- rep(FALSE, 4)
+     } else if (nnode == 5) {
+         is.subsamp <- c(FALSE, FALSE, FALSE, TRUE, FALSE)
+     } else stop("can only deal with graphs for individuals with 4 or 5 nodes",
+         "\nand graph is linear, and subsampling arrow is 4th of 5")
+     # fake object of class aster
+     randlab <- unlist(lapply(rout$random, colnames))
```

```
+     include.random <- grepl("paternalID", randlab, fixed = TRUE)
+     fake.out <- aout
+     fake.beta <- with(rout, c(alpha, b[include.random]))
+     modmat.random <- Reduce(cbind, random)
+     stopifnot(ncol(modmat.random) == length(rout$b))
+     # never forget drop = FALSE in programming R
+     modmat.random <- modmat.random[ , include.random, drop = FALSE]
+     fake.modmat <- cbind(fixed, modmat.random)
+     # now have to deal with objects of class aster (as opposed to reaster)
+     # thinking model matrices are three-way arrays.
+     stopifnot(prod(dim(aout$modmat)[1:2]) == nrow(fake.modmat))
+     fake.modmat <- array(as.vector(fake.modmat),
+         dim = c(dim(aout$modmat)[1:2], ncol(fake.modmat)))
+     fake.out$modmat <- fake.modmat
+     nparm <- length(rout$alpha) + length(rout$b) + length(rout$nu)
+     is.alpha <- 1:nparm %in% seq_along(rout$alpha)
+     is.bee <- 1:nparm %in% (length(rout$alpha) + seq_along(rout$b))
+     is.nu <- (! (is.alpha | is.bee))
+     # figure out individuals from each family
+     m <- rout$random$parental
+     dads <- grep("paternal", colnames(m))
+     # get family, that is, paternalID or grandpaternalID as the case may be
+     fams <- colnames(m)[dads] |> sub("^.*ID", "", x = _)
+     # drop maternal effects columns (if any)
+     m.dads <- m[ , dads, drop = FALSE]
+     # make into 3-dimensional array, like obj$modmat
+     m.dads <- array(m.dads, c(nind, nnode, ncol(m.dads)))
+     # only keep fitness node
+     # only works for linear graph
+     m.dads <- m.dads[ , nnode, ]
+     # redefine dads as families of individuals
+     stopifnot(as.vector(m.dads) %in% c(0, 1))
+     stopifnot(rowSums(m.dads) == 1)
+     # tricky, only works because each row of m.dads
+     # is indicator vector of family,
+     # so we are multiplying family number by zero or one
+     dads <- drop(m.dads %*% as.integer(fams))
+     # find one individual in each family
+     sudads <- sort(unique(dads))
+     which.ind <- match(sudads, dads)
```

```
+     function(alphabeenu) {
+         stopifnot(is.numeric(alphabeenu))
+         stopifnot(is.finite(alphabeenu))
+         stopifnot(length(alphabeenu) == nparm)
+         alpha <- alphabeenu[is.alpha]
+         bee <- alphabeenu[is.bee]
+         nu <- alphabeenu[is.nu]
+         fake.beta <- c(alpha, bee[include.random])
+         fake.out$coefficients <- fake.beta
+         pout <- predict(fake.out, model.type = "conditional",
+             is.always.parameter = TRUE)
+         xi <- matrix(pout, ncol = nnode)
+         xi <- xi[ , ! is.subsamp, drop = FALSE]
+         mu <- apply(xi, 1, prod)
+         mu <- mu[which.ind]
+         names(mu) <- paste0("PID",
+             formatC(sudads, format="d", width=3, flag="0"))
+         return(mu)
+     }
+ }
```

Rather than try to explain what this function does and how — see the supplementary material for Shaw, et al. (in preparation) for that — just notice

- this function is very complicated and not easy to differentiate, and
- this function did arise in a real application, so it cannot be avoided.

So try it out.

```
> alphabeenu <- with(rout, c(alpha, b, nu))
> map <- map.factory(rout)
> mu.hat <- map(alphabeenu)
> mu.hat

  PID001   PID006   PID008   PID015   PID016   PID028   PID034   PID036
1.990591 2.064271 1.111101 2.192411 1.549630 1.751216 3.070592 1.775515
  PID037   PID038   PID044   PID058   PID060   PID063   PID071   PID076
1.402446 1.914469 1.111345 3.140484 2.043843 2.398764 2.061942 1.586514
  PID081   PID098   PID099   PID106   PID110   PID112   PID115   PID119
2.175566 2.029489 1.603035 1.461212 2.067527 1.435715 2.433127 1.112868
```

```
  PID122   PID124   PID126   PID127   PID130   PID131   PID135   PID138
1.662109 1.343533 1.699121 3.219975 2.299833 2.548421 1.227020 2.231037
  PID139   PID149   PID152   PID160   PID162   PID163   PID166   PID167
2.826841 1.545078 3.211666 2.337268 1.689828 3.024124 1.855835 2.158382
  PID198   PID204
1.785939 2.757090
```

5.4 One Application of the Delta Method

Great! Now we want an (approximate) variance-covariance matrix for this (vector) estimate.

```
> jack <- jacobian(map, alphabeenu)
> Sigma.mu.hat <- jack %*% Sigma %*% t(jack)
```

So that is one application of the delta method.

5.5 Another Application of the Delta Method

But we actually wanted a function of these estimates.

```
> fitness_change <- function(mu) mean(mu * (mu / mean(mu) - 1))
> delta.fitness <- fitness_change(mu.hat)
> delta.fitness

[1] 0.1682873
```

And, of course, we need variance for this.

```
> jack <- jacobian(fitness_change, mu.hat)
> Sigma.delta.fit <- jack %*% Sigma.mu.hat %*% t(jack)
> Sigma.delta.fit

              [,1]
[1,] 0.0002177411
```

And standard error

```
> sqrt(drop(Sigma.delta.fit))

[1] 0.01475605
```

This agrees with Table 1 of the supplementary material for Shaw, et al. (in preparation).

6 Summary

It works and requires no calculus, so is less prone to mistakes and easier to explain.

References

Geyer, C. J. (2025). R package `aster`: Aster Models, version 1.3-6. https://cran.r-project.org/package=aster.

Geyer, C. J., Kulbaba, M. W., Sheth, S. N., Pain, R. E., Eckhart, V. M., and Shaw, R. G. (2022). Correction for Kulbaba et al. (2019). *Evolution*, **76**, 3074. doi:10.1111/evo.14607. Supplementary material, version 2.0.1. doi:10.5281/zenodo.7013098.

Geyer, C. J., Ridley, C. E., Latta, R. G., Etterson, J. R., and Shaw, R. G. (2013). Local Adaptation and genetic effects on fitness: Calculations for exponential family models with random effects. *Annals of Applied Statistics*, **7**, 1778–1795. doi:10.1214/13-AOAS653.

Geyer, C. J., and Shaw, R. G. (2010). Hypothesis tests and confidence intervals involving fitness landscapes fit by aster models. Technical Report No. 674 revised. School of Statistics, University of Minnesota. https://hdl.handle.net/11299/56328.

Geyer, C. J., Wagenius, S., and Shaw, R. G. (2005). Aster models for life history analysis. Technical Report No. 644. School of Statistics, University of Minnesota. http://hdl.handle.net/11299/199666.

Geyer, C. J., Wagenius, S., and Shaw, R. G. (2007). Aster models for life history analysis. *Biometrika*, **94**, 415–426. doi:10.1093/biomet/asm030.

Shaw, R. G., and C. J. Geyer (2010). Inferring fitness landscapes. *Evolution*, **64**, 2510–2520. doi:https://doi.org/10.1111/j.1558-5646.2010.01010.x.

Shaw, R. G., Geyer, C. J., Kulbaba, M. W., Sheth, S. N., Eckhart, V. M. and Pain, R. E. (in preparation). Realization of ongoing evolutionary adaptation in the field. Supplementary material (data analysis) for a draft paper `analysis/realized.pdf` in this git repository: https://github.com/cjgeyer/MeanFitness. Permanent repository: doi:10.5281/zenodo.17093288.