Correction for Kulbaba et al. (2019)

Charles J. Geyer and Ruth G. Shaw

August 02, 2022

Contents

1 Abstract

This corrects some calculations in Kulbaba et al. (2019) that inadvertently failed to correct for subsampling as described by Shaw et al. (2008b), Section 8 of Shaw et al. (2008a), and the Supplementary Material of Stanton-Geddes et al. (2012). While we were at it, we also changed "block" from a fixed effect to a random effect in order to simplify interpretation.

2 License

3 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.14.
- The version of the `aster` package used to make this document is 1.1.2.
- The version of the `numDeriv` package used to make this document is 2016.8.1.1.
- The version of the `kableExtra` package used to make this document is 1.3.4.

As far as we know, any fairly recent version of R or these packages will do for processing this Rmarkdown document. We are not using cutting edge features.

Attach packages.

```
library("aster")
library("numDeriv")
library("kableExtra")
```

4 Data

Get data (if not done already).

```
u <- paste0("https://raw.githubusercontent.com/mason-kulbaba/",
    "adaptive-capacity/master/VaW_W_analyses/CS/csdata.csv")
if (! file.exists("csdata.csv"))
    download.file(u, "csdata.csv")
u <- paste0("https://raw.githubusercontent.com/mason-kulbaba/",
    "adaptive-capacity/master/VaW_W_analyses/GC/gcdata.csv")
if (! file.exists("gcdata.csv"))
    download.file(u, "gcdata.csv")
u <- paste0("https://raw.githubusercontent.com/mason-kulbaba/",
    "adaptive-capacity/master/VaW_W_analyses/KW/kwdata.csv")
if (! file.exists("kwdata.csv"))
    download.file(u, "kwdata.csv")
```

5 Introduction

We do aster (Shaw et al. (2008b), Geyer et al. (2013)) analyses for an aster model with graph

$$1 \xrightarrow{\text{Ber}} \texttt{Germ} \xrightarrow{\text{Ber}} \texttt{flw} \xrightarrow{\text{Poi}} \texttt{total.pods} \xrightarrow{\text{samp}} \texttt{total.pods.collected} \xrightarrow{\text{Poi}} \texttt{totalseeds}$$

where the variables are

- `Germ` is germination indicator (0 = no, 1 = yes), conditionally Bernoulli.
- `flw` is survival to flowering (0 = no, 1 = yes), conditionally Bernoulli.
- `total.pods` is total number of pods produced, conditionally Poisson.
- `total.pods.collected` is number of pods collected, conditionally Bernoulli. The arrow leading to this node is a subsampling arrow. The number of pods collected is a random sample of the pods produced.
- `totalseeds` is total number of seeds counted from collected pods, conditionally Poisson.

Set graphical model description in R.

```
vars <- c("Germ", "flw", "total.pods", "total.pods.collected", "totalseeds")
pred <- c(0, 1, 2, 3, 4)
fam <- c(1, 1, 2, 1, 2)
```

6 Example Analysis

6.1 Determine Data to Analyze

We start by doing one example, but we do it in such a way that code can be reused for all analyses. The following two variables determine (entirely) the analysis to be done.

```
mydata <- "gcdata.csv"
myyear <- 2015
```

6.2 Read Data

Read in data.

```
dat <- read.csv(mydata)
```

6.3 Remove Parental ID Zero

Parental ID Zero is apparently “unknown” and hence should be removed from these analyses.

```
zeros <- dat$paternalID == 0 | dat$maternalID == 0
dat <- dat[! zeros, ]
```

6.4 Make Factor Variables

Show types of variables.

```
sapply(dat, class)
```

```
##          positionID          maternalID          paternalID
##           "integer"           "integer"           "integer"
##            position                 row               block
##           "numeric"           "integer"         "character"
##                Germ                 flw          total.pods
##           "integer"           "integer"           "integer"
## total.pods.collected           totalseeds               cohort
##            "integer"            "integer"          "character"
##                 year             seedpool
##            "integer"            "logical"
```

Some of these variables need to be `factor`.

```
dat <- transform(dat,
    maternalID = as.factor(maternalID),
    paternalID = as.factor(paternalID),
    block = as.factor(block))
```

6.5 Fix Data

Check subsampling is correct.

```
oopsie <- with(dat, total.pods.collected > total.pods)
```

The following should say `FALSE`.

```
any(oopsie)
```

```
## [1] TRUE
```

For this example, there is a problem to fix.

```
if (any(oopsie))
    dat[oopsie, ]
```

```
##      positionID maternalID paternalID position row block Germ flw total.pods
## 5966       4308        177        138       23 602    6B    1   1          1
##      total.pods.collected totalseeds     cohort year seedpool
## 5966                    6          0 greenhouse 2016       NA
```

So this data error has to be checked and corrected. But for now (to make immediate progress on computing) we just assume `total.pods` is correct and equal to `total.pods.collected` for this row of the data.

```
if (any(oopsie))
    dat[oopsie, "total.pods.collected"] <- dat[oopsie, "total.pods"]
```

6.6 Subset Data

Subset the data to get one part we want to analyze separately. Other parts are analyzed in the same way. Divide into year-specific files.

```
subdat <- subset(dat, year == myyear & cohort == "greenhouse")
```

Show that we did the subset correctly.

```
unique(subdat$year)
```

```
## [1] 2015
```

Drop unused levels.

```
subdat <- droplevels(subdat)
```

6.7 Reshape Data

Reshape data the way R function `aster` wants it.

```r
redata <- reshape(subdat, varying = list(vars), direction = "long",
    timevar = "varb", times = as.factor(vars), v.names = "resp")
```

Add indicator variable `fit` to indicate "fitness" nodes (in these data just one node). Also add `root` (in these data always equal to 1).

```r
redata <- transform(redata,
    fit = as.numeric(grepl("totalseeds", as.character(varb))),
    root = 1)
```

6.8 Random Effect Aster Model

In order to have the same variance for the random effects for sires and dams we have to do the following.

```r
modmat.sire <- model.matrix(~ 0 + fit:paternalID, redata)
modmat.dam <- model.matrix(~ 0 + fit:maternalID, redata)
head(colnames(modmat.sire))
```

```
## [1] "fit:paternalID1"  "fit:paternalID6"  "fit:paternalID8"  "fit:paternalID15"
## [5] "fit:paternalID16" "fit:paternalID28"
```

```r
head(colnames(modmat.dam))
```

```
## [1] "fit:maternalID2"  "fit:maternalID3"  "fit:maternalID4"  "fit:maternalID7"
## [5] "fit:maternalID10" "fit:maternalID11"
```

```r
modmat.siredam <- cbind(modmat.sire, modmat.dam)
```

Then we can fit the model.

```r
rout <- reaster(resp ~ fit + varb,
    list(parental=~0 + modmat.siredam, block = ~ 0 + fit:block),
    pred, fam, varb, id, root, data = redata)
```

And show the results.

```r
summary(rout)
```

```
##
## Call:
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)               -0.5611     0.1199  -4.680 2.87e-06 ***
## fit                        2.3311     0.1239  18.817  < 2e-16 ***
## varbGerm                  -1.7953     0.1897  -9.464  < 2e-16 ***
## varbtotal.pods             1.9259     0.1379  13.967  < 2e-16 ***
## varbtotal.pods.collected  -5.3312     0.1640 -32.510  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental  0.09229    0.01014   9.098     <2e-16 ***
## block     0.01052    0.01105   0.952       0.17
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

7 Fit More Models for Same Data

The first thing we need to do is save our model fit so we do not clobber it with a new model fit. First make an empty list.

```
save.rout <- list()
```

Then save the fit we have.

```
site.name <- substr(mydata, 1, 2)
fit.name <- paste0(site.name, myyear)
fit.name
```

```
## [1] "gc2015"
```

```
save.rout[[fit.name]] <- rout
```

Now we need to put stuff done above in a loop. In the chunk below, we reuse multiple code chunks above (this is not visible in the PDF output, you must look at the Rmd source file).

```
for (myyear in 2016:2017) {
subdat <- subset(dat, year == myyear & cohort == "greenhouse")
subdat <- droplevels(subdat)
redata <- reshape(subdat, varying = list(vars), direction = "long",
    timevar = "varb", times = as.factor(vars), v.names = "resp")
redata <- transform(redata,
    fit = as.numeric(grepl("totalseeds", as.character(varb))),
    root = 1)
modmat.sire <- model.matrix(~ 0 + fit:paternalID, redata)
modmat.dam <- model.matrix(~ 0 + fit:maternalID, redata)
head(colnames(modmat.sire))
head(colnames(modmat.dam))
modmat.siredam <- cbind(modmat.sire, modmat.dam)
rout <- reaster(resp ~ fit + varb,
    list(parental=~0 + modmat.siredam, block = ~ 0 + fit:block),
    pred, fam, varb, id, root, data = redata)
site.name <- substr(mydata, 1, 2)
fit.name <- paste0(site.name, myyear)
fit.name
save.rout[[fit.name]] <- rout
}
```

What have we got?

```
names(save.rout)
```

```
## [1] "gc2015" "gc2016" "gc2017"
```

8 Fit More Models for Different Data

Now we need to analyze the data in the following files.

```
more.data <- c("kwdata.csv", "csdata.csv")
```

```
## [[1]]
##                          kwdata.csv
## "ddec14f72c0c6240858ea19d622db0ca"
##
## [[2]]
##                          csdata.csv
## "7a752987ab2ae37e22b2392bf6cf58c5"
```

Now we need to do the same thing as in the preceding section, except more complicated because we need to wrap that in a loop that reads data from those files.

```
for (mydata in more.data) {
dat <- read.csv(mydata)
zeros <- dat$paternalID == 0 | dat$maternalID == 0
dat <- dat[! zeros, ]
dat <- transform(dat,
    maternalID = as.factor(maternalID),
    paternalID = as.factor(paternalID),
    block = as.factor(block))
oopsie <- with(dat, total.pods.collected > total.pods)
if (any(oopsie))
    dat[oopsie, "total.pods.collected"] <- dat[oopsie, "total.pods"]
for (myyear in 2015:2017) {
subdat <- subset(dat, year == myyear & cohort == "greenhouse")
subdat <- droplevels(subdat)
redata <- reshape(subdat, varying = list(vars), direction = "long",
    timevar = "varb", times = as.factor(vars), v.names = "resp")
redata <- transform(redata,
    fit = as.numeric(grepl("totalseeds", as.character(varb))),
    root = 1)
modmat.sire <- model.matrix(~ 0 + fit:paternalID, redata)
modmat.dam <- model.matrix(~ 0 + fit:maternalID, redata)
head(colnames(modmat.sire))
head(colnames(modmat.dam))
modmat.siredam <- cbind(modmat.sire, modmat.dam)
rout <- reaster(resp ~ fit + varb,
    list(parental=~0 + modmat.siredam, block = ~ 0 + fit:block),
    pred, fam, varb, id, root, data = redata)
site.name <- substr(mydata, 1, 2)
fit.name <- paste0(site.name, myyear)
fit.name
save.rout[[fit.name]] <- rout
}
}
```

9 Function to Map from Canonical to Mean Value Parameter

We use the map factory concept (use a function to make another function). This provides many benefits over defining the function we want directly. It allows us to hide information in the function we make with the factory. It allows us to easily remake the function whenever we need it. And the latter is important, because the map from the canonical parameter scale to the mean value parameter scale depends on the data we have fit (depends on the fixed effects).

```
map.factory <- function(rout, is.subsamp) {
    stopifnot(inherits(rout, "reaster"))
    stopifnot(is.logical(is.subsamp))
    aout <- rout$obj
    stopifnot(inherits(aout, "aster"))
    nnode <- ncol(aout$x)
    if(nnode != length(is.subsamp))
        stop("length(is.subsamp) not the number of nodes in the aster graph")
    alpha <- rout$alpha
    ifit <- which(names(alpha) == "fit")
    if (length(ifit) != 1)
        stop("no fixed effect named fit")
    # return map function
    function (b) {
        stopifnot(is.numeric(b))
        stopifnot(is.finite(b))
        stopifnot(length(b) == 1)
        alpha[ifit] <- alpha[ifit] + b
        xi <- predict(aout, newcoef = alpha,
            model.type = "conditional", is.always.parameter = TRUE)
        xi <- matrix(xi, ncol = nnode)
        # always use drop = FALSE unless you are sure you don't want that
        # here if we omit drop = FALSE and there is only one non-subsampling
        # node, the code will break (apply will give an error)
        xi <- xi[ , ! is.subsamp, drop = FALSE]
        mu <- apply(xi, 1, prod)
        # mu is unconditional mean values for model without subsampling
        # in this application all components mu are the same because no
        # covariates except varb, so just return only one
        mu[1]
    }
}
```

Here R function `map.factory` when invoked returns a function that maps from a possible value of a parental effect on the canonical parameter scale (what the model has) to the corresponding value on the mean value parameter scale. This depends on the model and the fixed effect parameters, which are taken from argument `rout`, and it corrects for subsampling, which is indicated by the argument `is.subsamp`.

Note that we are ignoring blocks. This is the same as setting the block effect to zero. This is not the actual effect of any actual block. Rather, it is something like the population mean of the population of blocks. Specifically, we are setting the block effect to be the mean of the distribution that the model says block effects have (mean zero normal). We acknowledge that the blocks are not a simple random sample of some population of blocks, but this is the usual way that probability theory is simplified for introductory statistics courses and also, commonly, in generalized linear models with random effects.

Caution: this function does not handle the case where more than one node of the graph contributes to `fit` nor does it check. It assumes we have a linear graph, which we do

```
identical(pred, seq(along = pred) - 1)
```

```
## [1] TRUE
```

(If we did not have a linear graph, then correction for subsampling would be much more complicated.)

Invoke the function to construct an R function `map` that maps from canonical to mean value parameter values for these data. Also vectorize this function, which is useful in certain contexts.

```
map <- map.factory(rout, vars == "total.pods.collected")
```

```
mapv <- Vectorize(map)
```

Plot this function.

```
sigma.hat <- rout$sigma["parental"]
curve(mapv, from = -3 * sigma.hat, to = 3 * sigma.hat, log = "y")
```

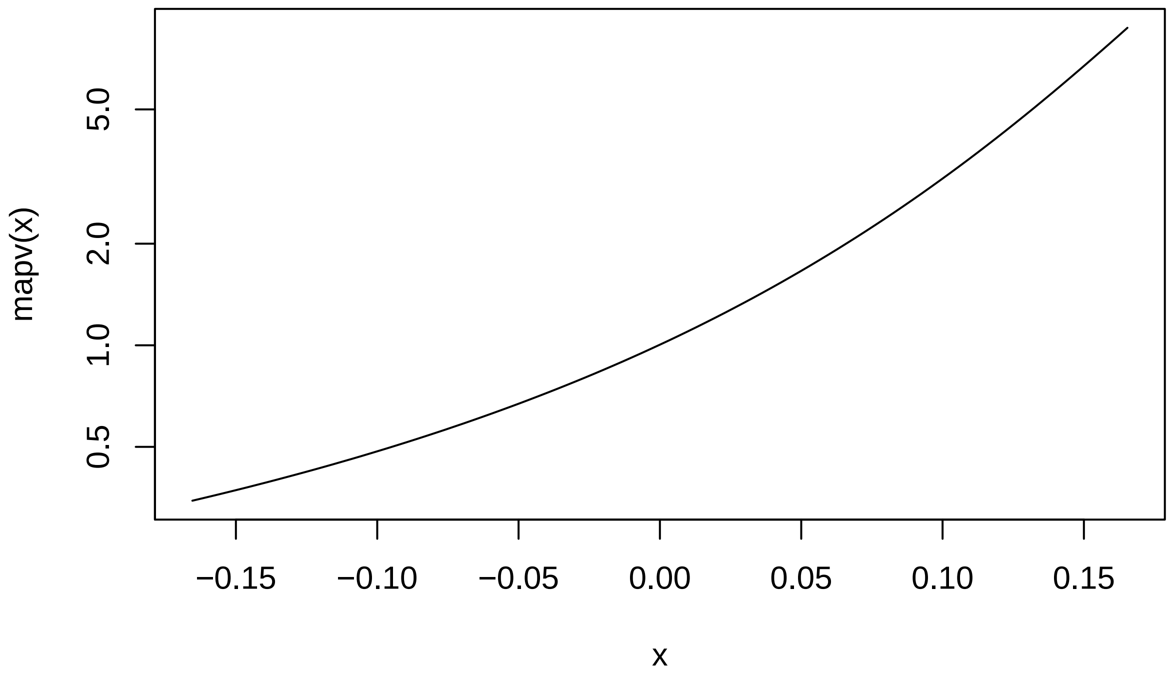

Figure 1: Map from parental random effect on the canonical parameter scale to the mean value parameter scale.

We digress to note that this map indicates a very large range of fitness on the mean value parameter scale (Fig. 1). This may seem surprising, but it results from the nonlinearity of the relationship between the canonical parameter scale and the mean value parameter scale and follows the conventional logic of generalized linear mixed models (GLMM) (Stiratelli et al. 1984) or aster models with random effects (Geyer et al. 2013). Admittedly, three standard deviations is a fairly extreme value for a normal random variable (happens with probability 0.0026998).
Similar issues affect all GLMM and aster models with random effects.

10 Distribution of Breeding Value Estimates

R function `reaster` provides estimates of the "breeding values" (in scare quotes) on the canonical parameter scale. Of course, these are not precisely estimates because breeding values are random effects rather than unknown parameters. So they are more analogous to BLUPS of breeding values in conventional quantitative genetics. Except they are not really that either due to the nonlinearity of the mapping from canonical to mean values shown above. If we replace BLUP by best median unbiased, then maybe we have some sort of analogy (because medians map to medians going through any monotone transformation).

Regardless, we map these quantities to the mean value parameter scale. Actually, rather than map all of the breeding values, we map a density estimate. We do this just for sire breeding values.

```
b <- rout$b
head(names(b))
```

```
## [1] "modmat.siredamfit:paternalID1"  "modmat.siredamfit:paternalID8"
## [3] "modmat.siredamfit:paternalID15" "modmat.siredamfit:paternalID21"
## [5] "modmat.siredamfit:paternalID24" "modmat.siredamfit:paternalID25"
```

```
idx <- grep("paternal", names(b))
b <- b[idx]
```

Fix up names so not so verbose.

```
names(b) <- sub("modmat.siredamfit:", "", names(b))
```

Now a density plot.

```
foo <- density(b, bw = "SJ")
```

```
plot(foo, main = "", xlab = "b")
```

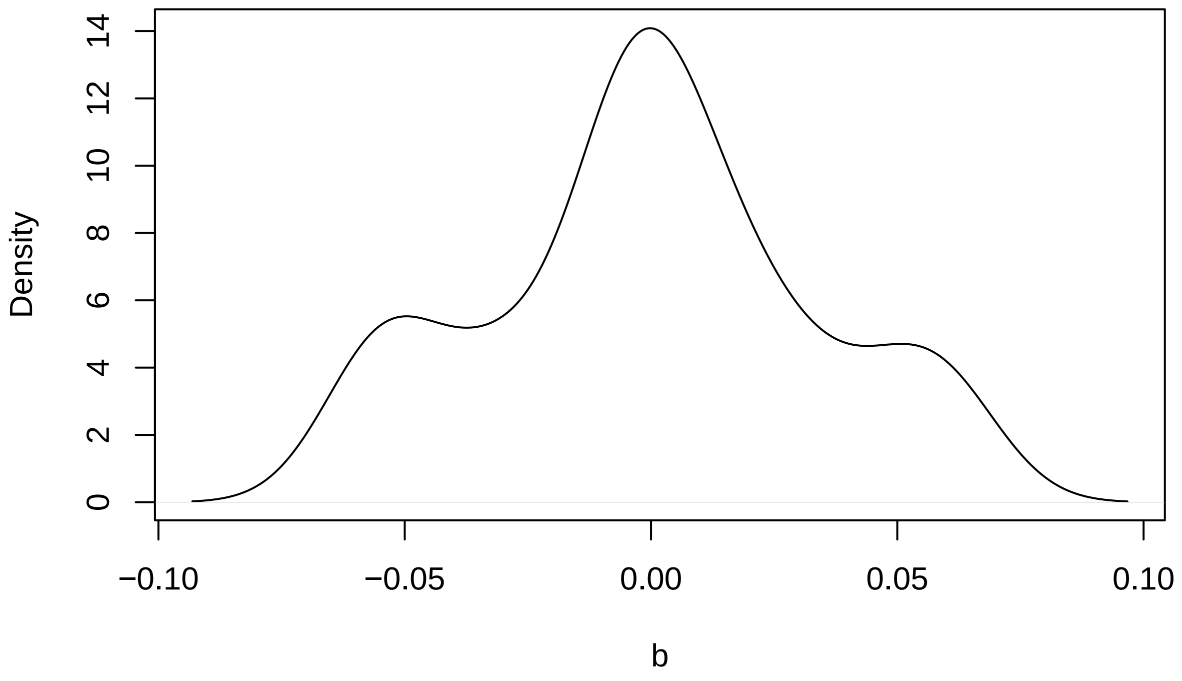

Figure 2: Density of estimated "breeding values" on the canonical parameter scale.

To move this density through a nonlinear one-to-one mapping, one must multiply by the derivative of the inverse mapping, which, by the inverse function theorem, is the same as dividing by the derivative of the mapping.

```r
bar <- foo
bar$x <- mapv(foo$x)
bar$y <- foo$y / grad(mapv, foo$x)
```

```r
plot(bar, main = "", xlab = "map(b)")
```

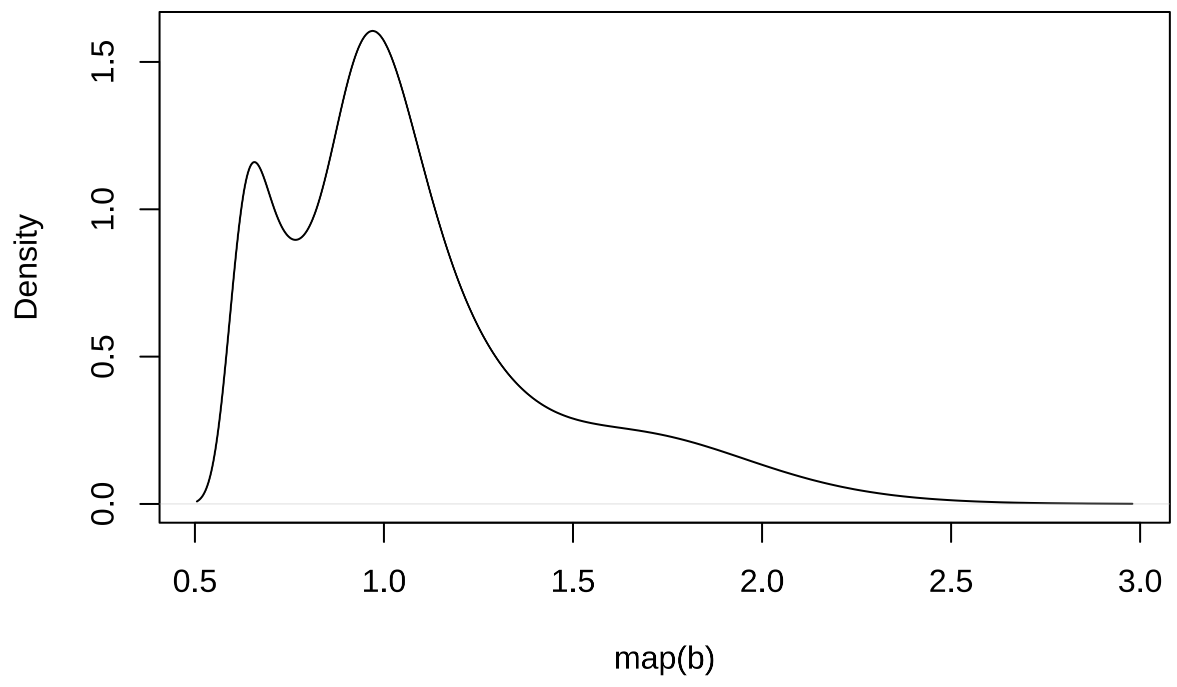

Figure 3: Density of estimated "breeding values" on the mean value parameter scale.

11 Distribution of Breeding Values for All Analyses

Now we repeat the process in the preceding section for all subsets of the data, except we stop before making the plots.

```r
fred <- function(rout) {
map <- map.factory(rout, vars == "total.pods.collected")

mapv <- Vectorize(map)
b <- rout$b
head(names(b))
idx <- grep("paternal", names(b))
b <- b[idx]
names(b) <- sub("modmat.siredamfit:", "", names(b))
foo <- density(b, bw = "SJ")
bar <- foo
bar$x <- mapv(foo$x)
bar$y <- foo$y / grad(mapv, foo$x)
return(bar)
}

density.all <- lapply(save.rout, fred)
```

Now we want to try to redo Figure 2 in Kulbaba et al. (2019). We want three plots, one for each site, and three curves on each plot, one for each year. And in order to use base R graphics, we need to find the maximum `x` value for each plot. Or perhaps the minimum `x` value such that the `y` value is below a specified cutoff above that `x`. And the cutoff should probably depend on the maximum `y` value, so find that first.

```
foo <- sapply(density.all, function(o) max(o$y))
ymax <- max(foo)
cutoff <- ymax / 100
foo <- sapply(density.all, function(o) max(o$x[o$y > cutoff]))
xmax <- max(foo)
```

Now we need to make the three plots (combined into one plot).

```
par(mar = c(5, 4, 4, 0) + 0.1, mfrow = c(1, 3))
plot(density.all[[1]], xlab = "breeding value (seed set)",
    xlim = c(0, xmax), ylim = c(0, ymax), main = "")
title(main = "Grey Cloud")
lines(density.all[[2]], col = "red")
lines(density.all[[3]], col = "blue")
legend(list(x = 4, y = 1.55), legend = as.character(2015:2017),
    lty = 1, col = c("black", "red", "blue"))
plot(density.all[[4]], xlab = "breeding value (seed set)",
    xlim = c(0, xmax), ylim = c(0, ymax), main = "")
title(main = "McCarthy Lake")
lines(density.all[[5]], col = "red")
lines(density.all[[6]], col = "blue")
legend(list(x = 4, y = 1.55), legend = as.character(2015:2017),
    lty = 1, col = c("black", "red", "blue"))
plot(density.all[[7]], xlab = "breeding value (seed set)",
    xlim = c(0, xmax), ylim = c(0, ymax), main = "")
title(main = "CERA")
lines(density.all[[8]], col = "red")
lines(density.all[[9]], col = "blue")
legend(list(x = 4, y = 1.55), legend = as.character(2015:2017),
    lty = 1, col = c("black", "red", "blue"))
```

12 Fisher's Fundamental Theorem of Natural Selection

12.1 Estimates Example

Here we calculate mean fitness, additive genetic variance for fitness, and their ratio, which Fisher's fundamental theorem of natural selection (FFTNS) says is the per generation change in mean fitness due to natural selection.

For this, following Geyer and Shaw (2013) we estimate mean fitness as

```
mf <- map(0)
mf
```

```
## [1] 1.004086
```

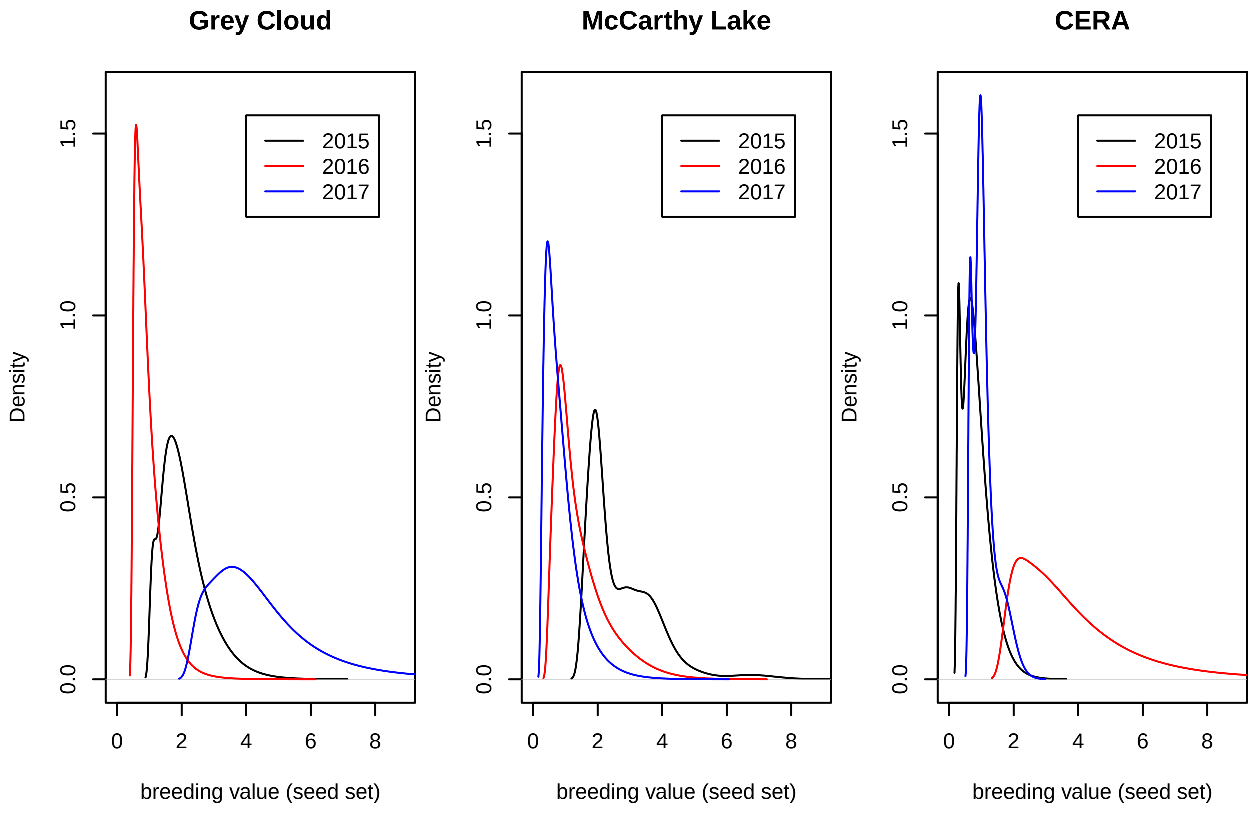

Figure 4: Density estimates of breeding values on mean value parameter scale.

and additive genetic variance for fitness as

```
vaw <- rout$sigma["parental"]^2 * grad(map, 0)^2
# get rid of name
vaw <- as.numeric(vaw)
vaw
```

```
## [1] 0.2473741
```

This is sire variance mapped to the mean value parameter scale.

In classical quantitative genetics the additive genetic variance is four times the contribution to it from sires, (Falconer and Mackay (1996), Chapter 9) and although it is unclear this applies outside of the classical models that assume the trait has a normal distribution (which we are not assuming here, instead using an aster model), we follow Geyer and Shaw (2013) in multiplying by 4.

```
vaw <- vaw * 4
vaw
```

```
## [1] 0.9894962
```

And now the Fisher's fundamental theorem calculation is

```
fftns <- vaw / mf
fftns
```

```
## [1] 0.9854696
```

12.2 Estimates for All Subsets of the Data

```
fran <- function(rout) {
map <- map.factory(rout, vars == "total.pods.collected")

mapv <- Vectorize(map)
mf <- map(0)
mf
vaw <- rout$sigma["parental"]^2 * grad(map, 0)^2
# get rid of name
vaw <- as.numeric(vaw)
vaw
vaw <- vaw * 4
vaw
fftns <- vaw / mf
fftns
return(list(mf = mf, vaw = vaw, fftns = fftns))
}

save.fftns <- lapply(save.rout, fran)
```

We have all three quantities (mean fitness, additive genetic variance for fitness, and the FFTNS prediction (their ratio)) saved, but we only make a table of additive genetic variance for fitness, following Kulbaba et al. (2019).

12.3 Standard Errors Example

12.3.1 Fisher Information

We are going to do standard errors via the delta method. For that we need derivatives, which we will again do using R package `numDeriv`. We also need Fisher information. We get an approximation for that from the method of R generic function `summary` for objects of class `reaster` which is also called `summary.reaster`

```r
sout <- summary(rout)
```

```r
names(sout)
```

```
## [1] "alpha"              "sigma"              "nu"
## [4] "object"             "standard.deviation" "fisher"
```

```r
length(rout$alpha)
```

```
## [1] 5
```

```r
length(rout$nu)
```

```
## [1] 2
```

```r
dim(sout$fisher)
```

```
## [1] 7 7
```

The value returned by `summary.reaster` is not documented. To understand that we have to Use the Source, Luke from which (not shown, but as for all R code, the source is right there, just type `summary.reaster` to see it, after `library(aster)` has been done) we see that

- component `fisher` of the result is indeed treated like the Fisher information matrix (it is just an approximation, not exact) which needs to be inverted to get the (approximate) asymptotic variance of the (approximate) maximum likelihood estimates (MLE) of the parameters,
- the parameters are the fixed effects `alpha` and the variance components `nu`, and
- the fixed effect parameters come first in the parameter vector and the variance components come last, that is `fisher` is the Fisher information matrix for the parameter `c(alpha, nu)`.

Then we get inverse Fisher information

```r
fishinv <- solve(sout$fisher)
```

12.3.2 Fisher's Fundamental Theorem Prediction

We need paper-and-pencil expressions for the derivatives of FFTNS prediction (as we shall see, we cannot use R function `grad` for this).

$$\begin{aligned}
\frac{\partial}{\partial \alpha} \frac{V_A(W)}{\mu} &= \frac{\partial}{\partial \alpha} 4 \mu^{-1} \nu \left[\left(\frac{\partial \mu}{\partial b} \right)_{b = 0} \right]^2 \\
&= - 4 \mu^{-2} \frac{\partial \mu}{\partial \alpha} \nu \left[\left(\frac{\partial \mu}{\partial b} \right)_{b = 0} \right]^2 + 8 \mu^{-1} \nu \left(\frac{\partial^2 \mu}{\partial b \partial \alpha} \right)_{b = 0} \\
\frac{\partial}{\partial \nu} \frac{V_A(W)}{\mu} &= \frac{\partial}{\partial \nu} 4 \mu^{-1} \nu \left[\left(\frac{\partial \mu}{\partial b} \right)_{b = 0} \right]^2 \\
&= 4 \mu^{-1} \left[\left(\frac{\partial \mu}{\partial b} \right)_{b = 0} \right]^2
\end{aligned}$$

where

- $$V_A(W) = 4 \nu \left[\left(\frac{\partial \mu}{\partial b} \right)_{b = 0} \right]^2$$
- μ is (predicted) mean fitness, what was `map(0)` before,

- b is the "breeding value (in scare quotes) on the canonical parameter scale, what was the argument of R function `map` before,
- α is the vector of fixed effects (`rout$alpha`),
- ν is the parental variance component (`rout$nu[1]` because we made that the first variance component, the second variance component being the one for blocks), and
  - $\partial\mu/\partial b$ is a scalar because b is a scalar,
  - $\partial\mu/\partial\alpha$ is a vector because α is a vector, and
  - $\partial^2\mu/\partial b\partial\alpha$ is a vector because b is a scalar and α is a vector.

We see that we need first derivatives of (predicted) mean fitness with respect to both breeding value ($\partial\mu/\partial b$) and fixed effects ($\partial\mu/\partial\alpha$) and need (mixed) second derivatives with respect to both of these ($\partial^2\mu/\partial b\partial\alpha$).

We could try to calculate second derivatives by applying R function `grad` in R package `numDeriv` twice, the second time to the result of calling it the first time, but this will not work well. R function `grad` expects a differentiable function that is as continuous as it can be given the inaccuracy of computer arithmetic. But R function `grad` does not return such. Instead we use R function `hessian` in R package `numDeriv` which is designed to calculate second derivatives. It does more than we want, but it does do what we want.

In order to be able to differentiate with respect to both b and α we need to rewrite R function `map` to be a function of both. But it cannot be a function of two arguments (not what R function `grad` expects) so we make it a function of one vector `balpha` that combines both.

```r
map.factory.too <- function(rout, is.subsamp) {
    stopifnot(inherits(rout, "reaster"))
    stopifnot(is.logical(is.subsamp))
    aout <- rout$obj
    stopifnot(inherits(aout, "aster"))
    nnode <- ncol(aout$x)
    if(nnode != length(is.subsamp))
        stop("length(is.subsamp) not the number of nodes in the aster graph")
    alpha <- rout$alpha
    ifit <- which(names(alpha) == "fit")
    if (length(ifit) != 1)
        stop("no fixed effect named fit")
    # return map function
    function (balpha) {
        stopifnot(is.numeric(balpha))
        stopifnot(is.finite(balpha))
        stopifnot(length(balpha) == 1 + length(alpha))
        b <- balpha[1]
        alpha <- balpha[-1]
        alpha[ifit] <- alpha[ifit] + b
        xi <- predict(aout, newcoef = alpha,
            model.type = "conditional", is.always.parameter = TRUE)
        xi <- matrix(xi, ncol = nnode)
        # always use drop = FALSE unless you are sure you don't want that
        # here if we omit drop = FALSE and there is only one non-subsampling
        # node, the code will break (apply will give an error)
        xi <- xi[ , ! is.subsamp, drop = FALSE]
        mu <- apply(xi, 1, prod)
        # mu is unconditional mean values for model without subsampling
        # in this application all components mu are the same because no
        # covariates except varb, so just return only one
        mu[1]
    }
}
```

And then make the function by invoking the factory function.

```
map.too <- map.factory.too(rout, vars == "total.pods.collected")
```

And then the point where we want to evaluate it.

```
balpha.hat <- c(0, rout$alpha)
```

And then check that the value of the function is what it is supposed to be.

```
all.equal(map(0), map.too(balpha.hat))
```

```
## [1] TRUE
```

So now we calculate both first and second derivatives of this function.

```
g <- grad(map.too, balpha.hat)
h <- hessian(map.too, balpha.hat)
```

And then we only keep the partial derivatives that occur in our formulae.

```
dmu.db <- g[1]
dmu.dalpha <- g[-1]
d2mu.db.dalpha <- h[1, -1]
```

And then we give names to the estimators in our formulas.

```
mu.hat <- map.too(balpha.hat)
nu.hat <- rout$nu["parental"]
```

And then we calculate the gradient vector of the FFTNS prediction with respect to the parameters of the model (fixed effects and variance components) using the formulae at the beginning of this section.

```
dfftns <- c(- 4 * nu.hat * dmu.dalpha * dmu.db^2 / mu.hat^2 +
    8 * nu.hat * d2mu.db.dalpha / mu.hat, 4 * dmu.db^2 / mu.hat, 0)
```

And apply the delta method.

```
fftns.se <- t(dfftns) %*% fishinv %*% dfftns
fftns.se <- sqrt(as.vector(fftns.se))
fftns.se
```

```
## [1] 0.4679352
```

12.3.3 Additive Genetic Variance for Fitness

The formulae here are a bit simpler than those in the preceding section.

$$\begin{aligned}\frac{\partial V_A(W)}{\partial \alpha} &= 8\nu \left(\frac{\partial^2 \mu}{\partial b \partial \alpha} \right)_{b=0} \\ \frac{\partial V_A(W)}{\partial \nu} &= 4 \left[\left(\frac{\partial \mu}{\partial b} \right)_{b=0} \right]^2\end{aligned}$$

So we calculate the gradient vector $V_A(W)$ with respect to the parameters of the model using these formulae.

```
dvaw <- c(8 * nu.hat * d2mu.db.dalpha, 4 * dmu.db^2, 0)
```

And then apply the delta method to get standard errors for this estimator.

```
vaw.se <- t(dvaw) %*% fishinv %*% dvaw
vaw.se <- sqrt(as.vector(vaw.se))
vaw.se
```

```
## [1] 0.3710482
```

12.3.4 Mean Fitness

Now the gradient vector of μ with respect to the parameters of the model is

```
dmf <- c(dmu.dalpha, 0, 0)
```

And then apply the delta method to get standard errors for this estimator.

```
mf.se <- t(dmf) %*% fishinv %*% dmf
mf.se <- sqrt(as.vector(mf.se))
mf.se
```

```
## [1] 0.3683383
```

12.3.5 Comment about Block Variance Component

It might occur to one to ask where the sampling distribution of the variance component for blocks enters. It does not appear in any of our formulas because none of the quantities we are estimating depends on it. But we have not left it out of our calculation because the (approximate) Fisher information matrix is for *all* estimated parameters so its inverse `fishinv` gives the asymptotic joint distribution for *all* parameters, and this does correctly take into account the sampling variability in the variance component for blocks.

12.3.6 Comment about the Bootstrap

This is not the way that Kulbaba et al. (2019) did confidence intervals for these parameters. They reported double parametric bootstrap t confidence intervals, which are not symmetric (so they do not have the form mean $\pm\, 1.96 \cdot$ standard error as would be our intervals (if we did report intervals rather than just standard errors). They needed a double bootstrap because they did not have the standard error formulas we derive here, and thus needed a single bootstrap to estimate standard errors, and then a bootstap of the bootstrap (double bootstrap) to do confidence intervals. Because the bootstrap is much more reliable than asymptotics, their intervals would have been better than ours except for omitting the correction for subsampling. Now, with the standard error formulas derived in this section available, one could do as well as one can do with single parametric bootstrap t confidence intervals. But we do not take the extra computing time to do that in this correction.

12.4 Standard Errors for All Subsets of the Data

```
sally <- function(rout) {
map.too <- map.factory.too(rout, vars == "total.pods.collected")
balpha.hat <- c(0, rout$alpha)
mu.hat <- map.too(balpha.hat)
nu.hat <- rout$nu["parental"]
g <- grad(map.too, balpha.hat)
h <- hessian(map.too, balpha.hat)
dmu.db <- g[1]
dmu.dalpha <- g[-1]
d2mu.db.dalpha <- h[1, -1]
```

```r
sout <- summary(rout)
fishinv <- try(solve(sout$fisher))
if (inherits(fishinv, "try-error"))
    return(list(mf = NA, vaw = NA, fftns = NA))
dfftns <- c(- 4 * nu.hat * dmu.dalpha * dmu.db^2 / mu.hat^2 +
    8 * nu.hat * d2mu.db.dalpha / mu.hat, 4 * dmu.db^2 / mu.hat, 0)
fftns.se <- t(dfftns) %*% fishinv %*% dfftns
fftns.se <- sqrt(as.vector(fftns.se))
fftns.se
dvaw <- c(8 * nu.hat * d2mu.db.dalpha, 4 * dmu.db^2, 0)
vaw.se <- t(dvaw) %*% fishinv %*% dvaw
vaw.se <- sqrt(as.vector(vaw.se))
vaw.se
dmf <- c(dmu.dalpha, 0, 0)
mf.se <- t(dmf) %*% fishinv %*% dmf
mf.se <- sqrt(as.vector(mf.se))
mf.se
return(list(mf = mf.se, vaw = vaw.se, fftns = fftns.se))
}

save.se <- lapply(save.rout, sally)
```

```
## Error in solve.default(sout$fisher) :
##   Lapack routine dgesv: system is exactly singular: U[5,5] = 0
## Error in solve.default(sout$fisher) :
##   Lapack routine dgesv: system is exactly singular: U[5,5] = 0
```

So now we are ready to redo Table 3 in Kulbaba et al. (2019).

```r
est <- unlist(save.fftns)
est <- formatC(est, digits = 3, format = "f")
se <- unlist(save.se)
se <- formatC(se, digits = 3, format = "f")
est.se <- paste0(est, " (", se, ")")
est.se <- matrix(est.se, ncol = 3, byrow = TRUE)
rownames(est.se) <- rep(2015:2017, times = 3)
colnames(est.se) <- c("$\\overline{W}$", "$V_A(W)$",
        "$V_A(W) / \\overline{W}$")

kbl(est.se,
    caption = "Estimates with Standard Errors (in parentheses)",
    format = "latex", escape = FALSE, booktabs = TRUE) %>%
    kable_styling() %>%
    pack_rows("Grey Cloud Dunes", 1, 3) %>%
    pack_rows("McCarthy Lake", 4, 6) %>%
    pack_rows("CERA", 7, 9)
```

In this table there are two rows in which we cannot calculate standard errors because the (approximate) Fisher information matrix is too close to being singular and hence cannot be inverted.

13 Breeding Values versus Breeding Values

This section is about reproducing Figure 3 in Kulbaba et al. (2019) except with the correction for subsampling.

First get all of the breeding values mapped to the canonical parameter scale.

Table 1: Estimates with Standard Errors (in parentheses)

	$\overline{W}$	$V_A(W)$	$V_A(W)/\overline{W}$
Grey Cloud Dunes			
2015	1.872 (0.239)	4.583 (0.963)	2.448 (0.634)
2016	0.839 (0.122)	0.736 (0.166)	0.878 (0.208)
2017	4.052 (0.552)	15.802 (3.701)	3.900 (1.136)
McCarthy Lake			
2015	2.546 (NA)	11.762 (NA)	4.620 (NA)
2016	1.230 (0.295)	4.118 (0.920)	3.347 (1.187)
2017	0.731 (0.150)	1.505 (0.331)	2.060 (0.630)
CERA			
2015	0.698 (NA)	1.484 (NA)	2.126 (NA)
2016	3.253 (0.550)	8.775 (2.470)	2.697 (0.903)
2017	1.004 (0.368)	0.989 (0.371)	0.985 (0.468)

```r
andrew <- function(rout) {
map <- map.factory(rout, vars == "total.pods.collected")

mapv <- Vectorize(map)
b <- rout$b
head(names(b))
idx <- grep("paternal", names(b))
b <- b[idx]
names(b) <- sub("modmat.siredamfit:", "", names(b))
return(mapv(b))
}

save.e <- lapply(save.rout, andrew)
```

Check that the names agree within site.

```r
identical(names(save.e$gc2015), names(save.e$gc2016))
```

```
## [1] TRUE
```

```r
identical(names(save.e$gc2015), names(save.e$gc2017))
```

```
## [1] TRUE
```

```r
identical(names(save.e$kw2015), names(save.e$kw2016))
```

```
## [1] TRUE
```

```r
identical(names(save.e$kw2015), names(save.e$kw2017))
```

```
## [1] TRUE
```

```r
identical(names(save.e$cs2015), names(save.e$cs2016))
```

```
## [1] TRUE
```

```r
identical(names(save.e$cs2015), names(save.e$cs2017))
```

```
## [1] TRUE
```

```
par(mfrow = c(3,3))
par(mar = c(5,4,0,0) + 0.1)
plot(save.e$gc2015, save.e$gc2016, xlab = "GC 2015", ylab = "GC 2016")
plot(save.e$gc2015, save.e$gc2017, xlab = "GC 2015", ylab = "GC 2017")
plot(save.e$gc2016, save.e$gc2017, xlab = "GC 2016", ylab = "GC 2017")
plot(save.e$kw2015, save.e$kw2016, xlab = "ML 2015", ylab = "ML 2016")
plot(save.e$kw2015, save.e$kw2017, xlab = "ML 2015", ylab = "ML 2017")
plot(save.e$kw2016, save.e$kw2017, xlab = "ML 2016", ylab = "ML 2017")
plot(save.e$cs2015, save.e$cs2016, xlab = "CERA 2015", ylab = "CERA 2016")
plot(save.e$cs2015, save.e$cs2017, xlab = "CERA 2015", ylab = "CERA 2017")
plot(save.e$cs2016, save.e$cs2017, xlab = "CERA 2016", ylab = "CERA 2017")
```

14 References

Falconer, D. S., and T. F. C. Mackay. 1996. fourth. Longman, Harlow, England.

Geyer, C. J., C. E. Ridley, R. G. Latta, J. R. Etterson, and R. G. Shaw. 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 R. G. Shaw. 2013. Aster models with random effects and additive genetic variance for fitness. School of Statistics, University of Minnesota.

Kulbaba, M. W., S. N. Sheth, R. E. Pain, V. M. Eckhart, and R. G. Shaw. 2019. Additive genetic variance for lifetime fitness and the capacity for adaptation in an annual plant. Evolution 73:1746–1758, doi: 10.1111/evo.13830.

Shaw, R. G., C. J. Geyer, S. Wagenius, H. H. Hangelbroek, and J. R. Etterson. 2008a. More supporting data analysis for “Unifying life history analysis for inference of fitness and population growth”. School of Statistics, University of Minnesota.

Shaw, R. G., C. J. Geyer, S. Wagenius, H. H. Hangelbroek, and J. R. Etterson. 2008b. Unifying life history analysis for inference of fitness and population growth. American Naturalist 172:E35–E47, doi: 10.1086/588063.

Stanton-Geddes, J., P. Tiffin, and R. G. Shaw. 2012. Role of climate and competitors in limiting fitness across range edges of an annual plant. Ecology 93:1604–1613, doi: 10.1890/11-1701.1.

Stiratelli, R., N. Laird, and J. H. Ware. 1984. Random-effects models for serial observations with binary response. Biometrics 40:961–971, doi: 10.2307/2531147.

15 Appendix

Show model fits.

```
for (i in seq(along = save.rout)) {
    cat("\n\n***************", names(save.rout)[i], "***************\n\n")
    print(summary(save.rout[[i]]))
}
```

```
##
##
## *************** gc2015 ***************
##
##
## Call:
```

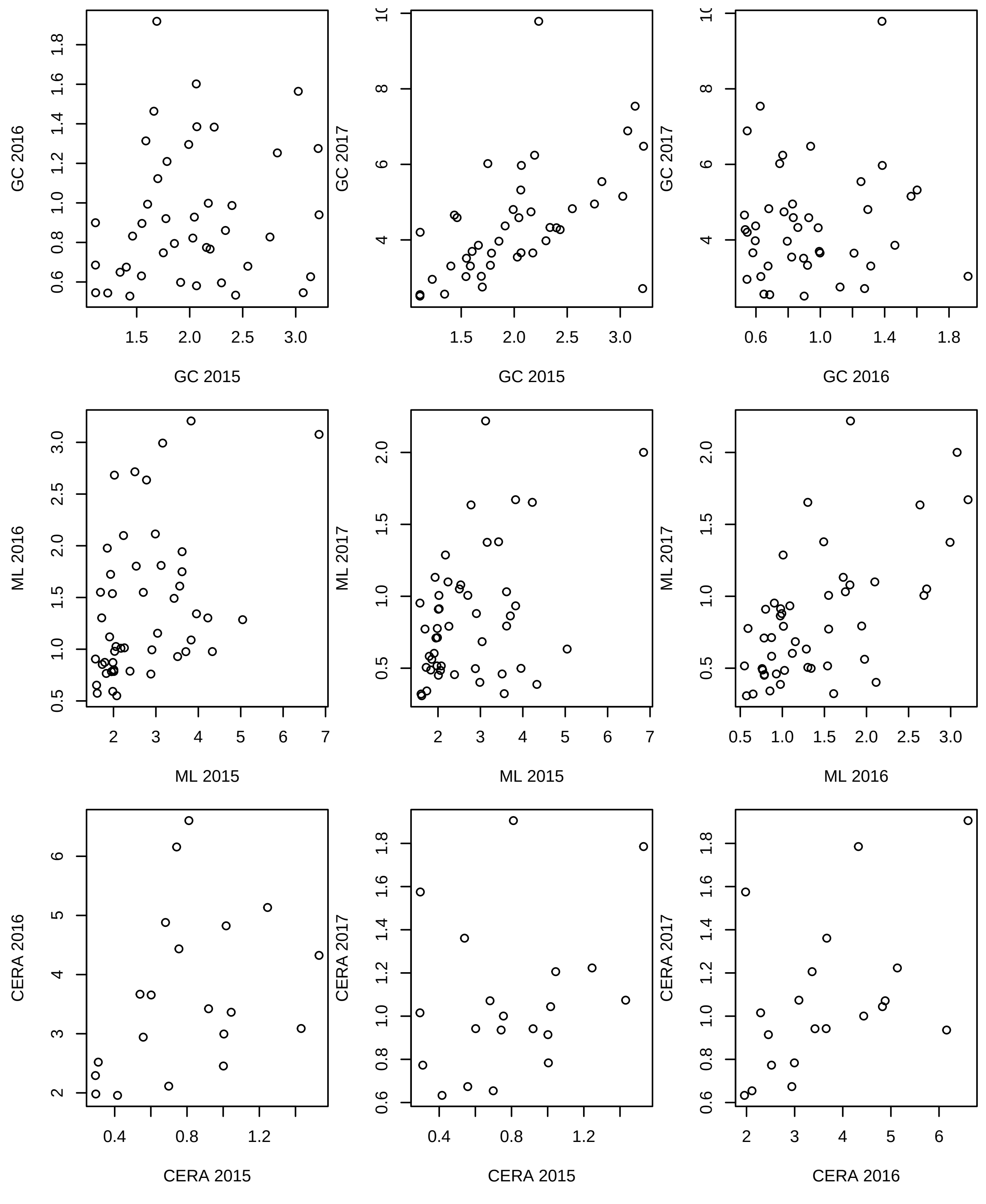

Figure 5: Comparing Breeding Values for Different Years, Same Site.

```
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)               -0.5611     0.1199  -4.680 2.87e-06 ***
## fit                        2.3311     0.1239  18.817  < 2e-16 ***
## varbGerm                  -1.7953     0.1897  -9.464  < 2e-16 ***
## varbtotal.pods             1.9259     0.1379  13.967  < 2e-16 ***
## varbtotal.pods.collected  -5.3312     0.1640 -32.510  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental  0.09229    0.01014   9.098     <2e-16 ***
## block     0.01052    0.01105   0.952       0.17
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** gc2016 ***************
##
##
## Call:
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)               1.16801    0.09632  12.127  < 2e-16 ***
## fit                       0.50168    0.10362   4.841 1.29e-06 ***
## varbGerm                 -3.29799    0.16515 -19.970  < 2e-16 ***
## varbtotal.pods           -0.97882    0.12283  -7.969 1.61e-15 ***
## varbtotal.pods.collected -5.85708    0.17027 -34.398  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental  0.10201    0.01171   8.713    < 2e-16 ***
## block     0.04627    0.01423   3.251   0.000575 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** gc2017 ***************
##
##
## Call:
```

```
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)               -4.4763     0.1485  -30.15   <2e-16 ***
## fit                        6.3364     0.1513   41.89   <2e-16 ***
## varbGerm                   2.3253     0.1985   11.71   <2e-16 ***
## varbtotal.pods             6.8270     0.1638   41.69   <2e-16 ***
## varbtotal.pods.collected  -2.6536     0.1838  -14.44   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental 0.065511   0.007521   8.710    < 2e-16 ***
## block    0.029992   0.008856   3.387   0.000354 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** kw2015 ***************
##
##
## Call:
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                            Estimate Std. Error z value Pr(>|z|)
## (Intercept)              -9.387e+00  4.064e-01  -23.10   <2e-16 ***
## fit                       1.160e+01  4.067e-01   28.52   <2e-16 ***
## varbGerm                  5.270e+00  5.259e-01   10.02   <2e-16 ***
## varbtotal.pods           -3.694e+01  8.389e+06    0.00        1
## varbtotal.pods.collected  5.072e+01  8.389e+06    0.00        1
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##           Estimate Std. Error z value Pr(>|z|)/2
## parental 0.0059749  0.0008864    6.74   7.89e-12 ***
## block    0.0000000         NA      NA         NA
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** kw2016 ***************
##
##
## Call:
```

```
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)               -1.6668     0.2008  -8.302  < 2e-16 ***
## fit                        3.6199     0.2022  17.905  < 2e-16 ***
## varbGerm                  -1.6825     0.3085  -5.454 4.93e-08 ***
## varbtotal.pods             0.6743     0.2624   2.569   0.0102 *
## varbtotal.pods.collected  -1.4371     0.2602  -5.524 3.32e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental 0.026245   0.002845   9.225    < 2e-16 ***
## block    0.013316   0.004252   3.131    0.00087 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** kw2017 ***************
##
##
## Call:
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)              -3.25893    0.20605 -15.816   <2e-16 ***
## fit                       4.13710    0.20906  19.789   <2e-16 ***
## varbGerm                 -0.06383    0.30888  -0.207    0.836
## varbtotal.pods            3.92823    0.22943  17.121   <2e-16 ***
## varbtotal.pods.collected  3.37681    0.21882  15.432   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental 0.063755   0.006991   9.120    < 2e-16 ***
## block    0.021956   0.008506   2.581    0.00492 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** cs2015 ***************
##
##
## Call:
```

```
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                           Estimate Std. Error z value Pr(>|z|)
## (Intercept)             -3.354e+00  2.105e-01 -15.936  < 2e-16 ***
## fit                      4.772e+00  2.166e-01  22.032  < 2e-16 ***
## varbGerm                 1.888e+00  2.513e-01   7.513 5.79e-14 ***
## varbtotal.pods          -3.764e+01  3.837e+05   0.000        1
## varbtotal.pods.collected  4.320e+01  3.837e+05   0.000        1
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental 0.045111   0.007059   6.391   8.24e-11 ***
## block    0.053274   0.023136   2.303     0.0106 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** cs2016 ***************
##
##
## Call:
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                         Estimate Std. Error z value Pr(>|z|)
## (Intercept)              -4.8466     0.1614 -30.028  < 2e-16 ***
## fit                       5.9962     0.1722  34.813  < 2e-16 ***
## varbGerm                  3.6948     0.2068  17.868  < 2e-16 ***
## varbtotal.pods            7.2322     0.1791  40.387  < 2e-16 ***
## varbtotal.pods.collected   0.7603     0.1892   4.018 5.87e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental  0.13089    0.01898   6.896   2.68e-12 ***
## block     0.05096    0.02391   2.131     0.0165 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## *************** cs2017 ***************
##
##
## Call:
```

```
## reaster.formula(fixed = resp ~ fit + varb, random = list(parental = ~0 +
##     modmat.siredam, block = ~0 + fit:block), pred = pred, fam = fam,
##     varvar = varb, idvar = id, root = root, data = redata)
##
##
## Fixed Effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)               -1.4799     0.2165  -6.835 8.19e-12 ***
## fit                        3.0650     0.2242  13.669  < 2e-16 ***
## varbGerm                  -1.0575     0.3110  -3.401 0.000672 ***
## varbtotal.pods             2.7616     0.2537  10.884  < 2e-16 ***
## varbtotal.pods.collected  -2.7238     0.2698 -10.095  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Square Roots of Variance Components (P-values are one-tailed):
##          Estimate Std. Error z value Pr(>|z|)/2
## parental  0.05514    0.01036   5.321   5.16e-08 ***
## block     0.07001    0.03054   2.292     0.0109 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```