Estimates of Realized Response to Selection in *Chamaecrista fasciculata* and Decomposition into Environmental and Genetic Parts

Charles J. Geyer* Mason W. Kulbaba† Seema N. Sheth‡ Rachel E. Pain§
Vincent M. Eckhart¶ Ruth G. Shaw‖

September 08, 2025

Contents

*School of Statistics, University of Minnesota, geyer@umn.edu, https://orcid.org/0000-0003-1471-1703
†St. Mary's University, mason.kulbaba@stmu.ca, https://orcid.org/0000-0003-0619-7089
‡Department of Plant and Microbial Biology, North Carolina State University, ssheth3@ncsu.edu, https://orcid.org/0000-0001-8284-7608
§Ecology, Evolution and Behavior Graduate Program, University of Minnesota, repain@umn.edu
¶Department of Biology, Grinnell College, eckhart@grinnell.edu
‖Department of Ecology, Evolution and Behavior, University of Minnesota, shawx016@umn.edu, https://orcid.org/0000-0001-5980-9291

Abstract

This work builds on Kulbaba *et al.* (2019) and the correction to it (Geyer *et al.*, 2022) to obtain estimates of the realized response to natural selection. Those articles presented estimates of mean fitness and additive genetic variance for fitness for three populations of *Chamaecrista fasciculata*, each grown in its home location in three years via aster analyses of records of components of fitness for a pedigreed set of individuals. Here, we consider the realized change in mean fitness from one generation to the next, for comparison with the prediction from Fisher's Fundamental Theorem of Natural Selection (FFTNS). We divide change in mean fitness in one generation into three parts: that due to change in genetic composition described by FFTNS (intragenerational change in mean additive genetic effects for fitness), that due to change in genetic composition not described by FFTNS (everything else at least partially genetic), and that due to change in environment. Here, we obtain estimates of a) mean fitness of the pedigreed parental populations before selection (previously presented in Kulbaba *et al.* (2019) and its correction); b) mean fitness of the pedigreed parental population after selection (i.e. accounting for the change in representation of the families reflected in differential seed production); and mean fitness of the offspring of the pedigreed sets (i. e., the outcome of natural selection on the parental generation when grown in the same sites in the following year).

We also obtain standard errors of our estimates. In this we use a new scheme that treats random effects as parameters to estimate because we do use estimates of random effects in our estimates of mean fitness.

1 License

The R markdown source for this document is the file `realized.Rmd` in the GitHub private repository https://github.com/cjgeyer/mf which will be made public whenever a paper based on it is submitted.

2 R

- The version of R used to make this document is 4.5.1.
- The version of the `rmarkdown` package used to make this document is 2.29.
- The version of the `bookdown` package used to make this document is 0.43.
- The version of the `aster` package used to make this document is 1.3.5.
- The version of the `numDeriv` package used to make this document is 2016.8.1.1.
- The version of the `Matrix` package used to make this document is 1.7.3.
- The version of the `parallel` package used to make this document is 4.5.1.
- The version of the `kableExtra` package used to make this document is 1.4.0.
- The version of the `Hmisc` package used to make this document is 5.2.3.
- The version of the `data.table` package used to make this document is 1.17.8.
- The version of the `flextable` package used to make this document is 0.9.9.
- The version of the `mcmc` package used to make this document is 0.9.8.

Attach packages.

```
library("aster")
library("numDeriv")
library("Matrix")
library("parallel")
options("mc.cores" = detectCores())
library("kableExtra")
suppressMessages(library("Hmisc"))
suppressMessages(library("flextable"))
library("data.table")
library("mcmc")
```

Need at least version 1.3-4 of R package `aster` for R generic function `vcov` to work on results of calls to R functions `aster` and `reaster`.

```
stopifnot(compareVersion(as.character(packageVersion("aster")), "1.3-4") >= 0)
```

This version is now on CRAN (https://cran.r-project.org/package=aster) so can be downloaded using R function `install.packages` or the equivalent by mousing around in the menus of some app.

A later version 1.3-5 was required by CRAN to fix some test output, but did not affect any functions in the package or their documentation. So we do not need that version. It was for CRAN's benefit only.

3 Data

3.1 Files

For the analyses here. the data files are

```
load("mf.rda")
ls()
```

```
## [1] "data.primary" "key.data"
```

```
sapply(data.primary, class)
```

```
##           CS           GC           KW
## "data.frame" "data.frame" "data.frame"
```

for

- Conard Environmental Research Area (CERA),
- Grey Cloud Dunes Scientific and Natural Area, and
- Kellogg-Weaver Dunes, also called McCarthy Lake,

respectively. These files include the same data on the same individuals as in the data files used by Kulbaba *et al.* (2019) and Geyer *et al.* (2022) but also include more individuals, who are offspring of those analyzed before. For more details, see Kulbaba *et al.* (2019).

Much preprocessing of these data has already been done. See the file `fixup-data.pdf` in this repository.

3.2 Structure

We do aster analyses with random effects (Shaw *et al.* (2008), Geyer *et al.* (2013)) 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 (i.e. each pod may be collected or not). 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.

As always with aster models, the name of the distribution for an arrow is the name of the conditional distribution of the successor variable given the predecessor variable. The arrow labeled `samp` is a subsampling arrow. It is a Bernoulli arrow but the sampling is experimental rather than biological. This arrow may be missing in some analyses (Section 3.3 below).

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)
```

3.3 Alternative Structure

Get years.

```
years <- sort(unique(data.primary[[1]]$year))
names(years) <- years
years
```

```
## 2015 2016 2017
## 2015 2016 2017
```

```
years.offspring <- with(data.primary[[1]], year[cohort == "field"]) |>
    sort() |> unique()
names(years.offspring) <- years.offspring
years.offspring
```

```
## 2016 2017
## 2016 2017
```

And for each site-year combination, subset data for parents (`cohort == "greenhouse"`) and check for presence of subsampling.

```
samp.parents <- lapply(data.primary, function(x) lapply(years, function(y) {
    subdat <- subset(x, year == y & cohort == "greenhouse")
    subdat <- droplevels(subdat)
    with(subdat, any(total.pods > total.pods.collected))
    }))
samp.parents
```

```
## $CS
## $CS$`2015`
## [1] FALSE
##
## $CS$`2016`
## [1] TRUE
##
## $CS$`2017`
## [1] TRUE
##
##
## $GC
## $GC$`2015`
## [1] TRUE
##
## $GC$`2016`
## [1] TRUE
##
## $GC$`2017`
## [1] TRUE
##
##
## $KW
## $KW$`2015`
## [1] FALSE
##
## $KW$`2016`
## [1] TRUE
```

```
##
## $KW$`2017`
## [1] TRUE
```

We see that we have two parental generation analyses (out of nine) in which there is no subsampling. For these we have to change the aster graph for individuals to omit the subsampling arrow.

```
vars.no.samp <- c("Germ", "flw", "total.pods", "totalseeds")
pred.no.samp <- c(0, 1, 2, 3)
fam.no.samp <- c(1, 1, 2, 2)
```

Now do the same check for offspring (`cohort == "field"`).

```
samp.offspring <- lapply(data.primary, function(x)
    lapply(years.offspring, function(y) {
    subdat <- subset(x, year == y & cohort == "field")
    subdat <- droplevels(subdat)
    with(subdat, any(total.pods > total.pods.collected))
    }))
samp.offspring
```

```
## $CS
## $CS$`2016`
## [1] TRUE
##
## $CS$`2017`
## [1] TRUE
##
##
## $GC
## $GC$`2016`
## [1] TRUE
##
## $GC$`2017`
## [1] TRUE
##
##
## $KW
## $KW$`2016`
## [1] TRUE
##
## $KW$`2017`
## [1] TRUE
```

So no issues with this for offspring: all offspring generation analyses have subsampling.

4 Analyses with R Function Reaster

4.1 Parents

We have nine analyses to do here, one for each site-year combination.

We have three covariates: sire, dam, and block, which we make random effects. We `cbind` the model matrices for sire and dam, so they share a variance component.

```
rout.parents <- mclapply(data.primary, function(x) mclapply(years, function(y) {
    subdat <- subset(x, year == y & cohort == "greenhouse")
```

```
    subdat <- droplevels(subdat)
    has.subsamp <- with(subdat, any(total.pods > total.pods.collected))
    if (has.subsamp) {
        redata <- reshape(subdat, varying = list(vars), direction = "long",
            timevar = "varb", times = as.factor(vars), v.names = "resp")
    } else {
        redata <- reshape(subdat, varying = list(vars.no.samp),
            direction = "long", timevar = "varb",
            times = as.factor(vars.no.samp), 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)
    modmat.siredam <- cbind(modmat.sire, modmat.dam)
    if (has.subsamp) {
        reaster(resp ~ fit + varb,
            list(parental = ~ 0 + modmat.siredam, block = ~ 0 + fit:block),
            pred, fam, varb, id, root, data = redata)
    } else {
        reaster(resp ~ fit + varb,
            list(parental = ~ 0 + modmat.siredam, block = ~ 0 + fit:block),
            pred.no.samp, fam.no.samp, varb, id, root, data = redata)
    }
    }, mc.preschedule = FALSE), mc.preschedule = FALSE)
```

Let's see what we got. Check what we got.

```
lapply(rout.parents, function(x) sapply(x, function(x) inherits(x, "reaster")))
```

```
## $CS
## 2015 2016 2017
## TRUE TRUE TRUE
##
## $GC
## 2015 2016 2017
## TRUE TRUE TRUE
##
## $KW
## 2015 2016 2017
## TRUE TRUE TRUE
```

Any variance components estimated to be zero?

```
lapply(rout.parents, function(x) sapply(x, function(x) x$nu))
```

```
## $CS
##                 2015       2016        2017
## parental 0.002034936 0.01713327 0.003040156
## block    0.002838072 0.00259683 0.004901275
##
## $GC
##                   2015        2016         2017
## parental 0.0085174772 0.010707779 0.0042917004
## block    0.0001106382 0.002180593 0.0008995002
```

```
##
## $KW
##                  2015         2016         2017
## parental 3.569967e-05 0.0006887975 0.0040702343
## block    0.000000e+00 0.0001773207 0.0004828604
```

One case where block effects are estimated to be zero, but all cases have parental effects estimated to be nonzero, which justifies further consideration of genetic effects.

4.2 Offspring

We have six analyses to do here, one for each site-year combination (in which there are offspring).

```
rout.offspring <- mclapply(data.primary,
    function(x) mclapply(years.offspring, function(y) {
    subdat <- subset(x, year == y & cohort == "field")
    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)
    reaster(resp ~ fit + varb,
        list(parental = ~ 0 + fit:grandpaternalID, block = ~ 0 + fit:block),
        pred, fam, varb, id, root, data = redata)
    }))
```

Let's see what we got. Check what we got.

```
lapply(rout.offspring, function(x)
    sapply(x, function(x) inherits(x, "reaster")))
```

```
## $CS
## 2016 2017
## TRUE TRUE
##
## $GC
## 2016 2017
## TRUE TRUE
##
## $KW
## 2016 2017
## TRUE TRUE
```

Any variance components estimated to be zero?

```
lapply(rout.offspring, function(x) sapply(x, function(x) x$nu))
```

```
## $CS
##                2016        2017
## parental 0.03575501 0.009716053
## block    0.01478358 0.001424909
##
## $GC
##                  2016         2017
## parental 0.0144848796 6.983372e-04
## block    0.0003548277 7.117278e-05
```

```
##
## $KW
##                    2016        2017
## parental 0.0009830560 0.003161333
## block    0.0002210912 0.000980789
```

No case where any variance components are estimated to be zero.

5 Asymptotic Variance-Covariance Matrices of Estimates

5.1 Parents

```
vcov.parents <- mclapply(rout.parents, function(x)
    mclapply(x, vcov, complete = TRUE, re.too = TRUE,
    standard.deviation = FALSE))
```

5.2 Offspring

```
vcov.offspring <- mclapply(rout.offspring, function(x)
    mclapply(x, vcov, complete = TRUE, re.too = TRUE,
    standard.deviation = FALSE))
```

6 Mapping Sire and Grandsire Effects to Mean Values

6.1 A Function Factory

We follow Section 9 of Geyer *et al.* (2022) *mutatis mutandis*. The main changes are here we will have a vectorizing function that simultaneously does mean fitness values for a specified set of individuals.

We follow the dictates of literate programming (Wikipedia page) developing our function a little bit at a time and then assembling all the bits into the whole function. The function we are developing will have one argument `rout`, so we make an object of the correct type to exercise our code bits on.

```
rout <- rout.parents[[1]][[1]]
```

Our first bit of code does some error checking and setup.

```
    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
```

The next job is to figure out the subsampling nodes of the graph (if present). Here the answer is the fourth of five.

```
    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")
```

Now we have to deal with the problem that the only R function we have that maps from canonical to mean value parameters is R function `predict.aster` and we need to give it an object of class `"aster"`. So we make one "by hand" rather than by invocation of R function `aster`. The simplest way to do that is to start with an already existing object of class `"aster"` that was produced by R function `aster` (which is found in our object of class `"reaster"`) and modify it as needed. Here we only want the random effects that have the string `"paternalID"` in their names (that is, either paternalID or grandpaternalID).

```
# 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
```

The next job is to figure out which parameters are which.

```
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))
```

The next job is to figure out which individuals are in which "families" (paternalID) and to get one of each for the output vector.

```
# 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)
```

That finishes the setup. Now we write the result of our factory function, which is another function with a single argument `alphabeenu` which is the vector of all the variables (fixed effects, random effects, and variance components.

```
alphabeenu <- with(rout, c(alpha, b, nu))
```

So first check that this argument is OK.

```
        stopifnot(is.numeric(alphabeenu))
        stopifnot(is.finite(alphabeenu))
        stopifnot(length(alphabeenu) == nparm)
```

Then extract the various parts. And make the `coefficients` component of our fake object of class `"aster"` to be the fixed effects vector `alpha` plus the random effects vector of the random effects we are "predicting" (those indicated by the indicator vector `include.random`).

```
        alpha <- alphabeenu[is.alpha]
        bee <- alphabeenu[is.bee]
        nu <- alphabeenu[is.nu]
        fake.beta <- c(alpha, bee[include.random])
        fake.out$coefficients <- fake.beta
```

Do the prediction, mapping unconditional submodel canonical parameters β to conditional mean value parameters ξ.

```
        pout <- predict(fake.out, model.type = "conditional",
            is.always.parameter = TRUE)
```

Then put the result ξ into matrix form, rows are individuals, columns are nodes of the graph for individuals. Toss the column of subsampling nodes (if any) and multiply the rest to get unconditional mean value parameters for fitness (mean fitness). Put in a comment that this only works for this particular graph.

```
        xi <- matrix(pout, ncol = nnode)
        xi <- xi[ , ! is.subsamp, drop = FALSE]
        mu <- apply(xi, 1, prod)
```

Finally, we put names on `mu` and return only one for each "family" (paternalID).

```
        mu <- mu[which.ind]
        names(mu) <- paste0("PID",
            formatC(sudads, format="d", width=3, flag="0"))
```

Now put it all together in a function factory.

```
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)
    }
}
```

R function `map.factory` has one argument, an object of class `reaster` produced by a call to R function `reaster` and produces a function with one argument, which is a vector $\theta = (\alpha, b, \nu)$ that contains the variables (fixed effects, standardized random effects, and standard deviation components), and that produced function returns mean fitness estimates for one individual from each "family" (sire for parental fits, grandsire for offspring fits). That is, it is a function whose value is another function.

Warning: This function only works for linear aster graphs for "individuals" (in scare quotes) having four or five arrows and the only subsampling is the fourth of five.

Try it out.

```
map <- map.factory(rout)
map(alphabeenu)
```

```
##    PID001    PID008    PID015    PID021    PID024    PID025    PID027    PID031
## 1.2456326 1.0443622 0.7426189 1.0038344 0.7552695 0.9192600 0.8104126 0.5400923
##    PID035    PID042    PID043    PID052    PID055    PID061    PID067    PID069
## 0.2936839 0.6810895 0.4158129 1.4312614 0.6017871 1.5302312 1.0164173 0.3092367
##    PID075    PID089    PID096    PID098
## 1.0013889 0.2955316 0.6988665 0.5578405
```

6.2 Apply To Parents

In hindsight, we did not want to apply R generic function `vcov` as we did above to make a separate list. (We could use R function `mapply` to work on multiple parallel lists, but don't choose to.) So we invoke R function `vcov` again here, putting estimates and their variance-covariance matrices in the same list.

```
moo.parents <- mclapply(rout.parents, function(x) mclapply(x, function(y) {
    map <- map.factory(y)
    alphabeenu <- with(y, c(alpha, b, nu))
    estimates <- map(alphabeenu)
    jack <- jacobian(map, alphabeenu)
    vcov.base <- vcov(y, complete = TRUE, re.too = TRUE,
        standard.deviation = FALSE)
    vcov.estimates <- jack %*% vcov.base %*% t(jack)
    return(list(estimates = estimates, vcov = vcov.estimates))
}))
```

In the above code we are applying the delta method for differentiable functions of asymptotically normal estimators. If $\hat{\theta}$ is asymptotically normal with variance given by R function `vcov` and g is a differentiable function, then the asymptotic variance of $g(\hat{\theta})$ is given by

$$J \operatorname{vcov}(\hat{\theta}) J^T$$

which is computed by the line

```
    vcov.estimates <- jack %*% vcov.base %*% t(jack)
```

in the code above. And we do not calculate the derivative matrix, also called Jacobian matrix, by calculus, but rather let the computer do it for us (numerically) using R function `jacobian` from R package `numDeriv`.

If you want to replace the function being used here (the g we are talking about is here implemented by R function `map`) by some other function, just do it. The delta method is valid for any differentiable function.

You replace R function `map` in both places it occurs (in computing R vector `estimates` and in computing R matrix `jack`) with your R function that calculates some other estimator.

6.3 Apply To Offspring

Same for offspring.

```
moo.offspring <- mclapply(rout.offspring, function(x) mclapply(x, function(y) {
    map <- map.factory(y)
    alphabeenu <- with(y, c(alpha, b, nu))
    estimates <- map(alphabeenu)
    jack <- jacobian(map, alphabeenu)
    vcov.base <- vcov(y, complete = TRUE, re.too = TRUE,
        standard.deviation = FALSE)
    vcov.estimates <- jack %*% vcov.base %*% t(jack)
    return(list(estimates = estimates, vcov = vcov.estimates))
}))
```

7 Genetic Change in Mean Fitness Due to Selection

7.1 Definition

One of the primary quantities of scientific interest, and the other quantities we calculate are related to it, is the part of the change in mean fitness in one generation that Fisher's fundamental theorem (FFTNS) addresses.

This quantity is very simple in some respects and a bit tricky in other respects. We start with the "breeder's equation" (also called the Robertson-Price equation). What does natural selection do? It changes frequencies of genotypes (or phenotypes) from what they were before selection to the same thing multiplied by fitness normalized to be a probability vector, that is, if θ are the frequencies before selection and μ is (expected) fitness, then

$$\theta \cdot \frac{\mu}{\operatorname{sum}(\mu)}$$

are the frequencies after selection. And the change in fitness is

$$\theta \cdot \frac{\mu}{\operatorname{sum}(\mu)} - \theta$$

and the change in mean fitness is the mean of the above

$$\operatorname{sum}\left(\theta \cdot \frac{\mu}{\operatorname{sum}(\mu)}\right) - \operatorname{mean}(\theta) \tag{1}$$

which can also be written

$$\operatorname{mean}\left(\theta \cdot \frac{\mu}{\operatorname{mean}(\mu)}\right) - \operatorname{mean}(\theta) \tag{2}$$

(the difference between `sum` and `mean` is a factor of `n` (the length of the vectors θ and μ) and this cancels in the numerator and denominator in going from one to the other.

7.2 Equivalent Definitions

This does not look like all treatments of the breeder's equation. Often one sees the covariance operator used. For any random variables X and Y

$$\operatorname{cov}(X, Y) = E(XY) - E(X)E(Y)$$

and in the special case $E(Y) = 1$ this becomes

$$\operatorname{cov}(X, Y) = E(XY) - E(X)$$

and in (2) we have arranged that the random variable $Y = \mu/\text{mean}(\mu)$ does have mean one by the simple expedient of dividing μ by its mean. Thus (2) is equivalent to

$$\operatorname{cov}\left(\theta, \frac{\mu}{\text{mean}(\mu)}\right) \tag{3}$$

This is shorter, but we think it obscures the logic. For those who like their math mysterious, (3) is better.

Equation (3) can be made even more mysterious by introducing the term *relative fitness* for $\mu/\,\text{mean}(\mu)$ so it becomes

$$\operatorname{cov}\left(\text{trait}, \text{relative fitness}\right) \tag{4}$$

7.3 Weighted Averages

We think it is less obscure to go back to (1) and introduce the notion of *weighted average*. A *weighted average* of a vector θ is

$$\sum_i \theta_i p_i \tag{5}$$

where p is a probability vector, that is, its components p_i are nonnegative and sum to one. A simple average is the special case where $p_i = 1/n$ for all i where n is the length of θ and p.

Another helpful concept is that of *unnormalized weights*. If μ is a vector with nonnegative components, then $\mu/\,\text{sum}(\mu)$ is a probability vector. When we use this probability vector to make a weighted average, we call μ the *unnormalized* weight vector and $\mu/\,\text{sum}(\mu)$ the *normalized* weight vector.

So (1) is just the difference between a weighted average of θ and a simple average of θ with μ as the unnormalized weight vector.

7.4 The Tricky Bit

Now we want to apply this logic where the trait θ under discussion is fitness itself. So we set $\theta = \mu$ giving

$$\text{mean}\left(\mu \cdot \frac{\mu}{\text{mean}(\mu)}\right) - \text{mean}(\mu) \tag{6}$$

7.5 Too Long, Didn't Read

Whether one regards all of the above as trivial, deep, clear, or confusing, it is the only bit of math we know of that addresses the question. Lande and Arnold (1983) discuss three different ways to express the same concept under certain assumptions (which are further discussed in Geyer and Shaw (2008)) but this does not change the concept that (2) expresses. Or (6) expresses.

In short, for each pedigreed cohort, we have obtained a vector μ returned by R function `map`, which estimates family-specific mean absolute fitnesses. To estimate overall mean fitness, we obtain a weighted sum of these μ, using as weights, the relative fitness of each family, obtained by dividing μ by mean(μ). We obtain the change in mean fitness by subtracting the mean fitness before selection, when families were equally represented.

7.6 Application

We make this into a function so we can differentiate it numerically.

```
fitness_change <- function(mu) mean(mu * (mu / mean(mu) - 1))
```

7.6.1 Parents

Now we apply this function to parents

```
change.parents <- mclapply(moo.parents, function(x) mclapply(x, function(y) {
    grad <- grad(fitness_change, y$estimates)
    my.vcov <- t(grad) %*% y$vcov %*% grad
    return(list(estimates = fitness_change(y$estimates), vcov = my.vcov))
}))
```

Here we do the delta method slightly differently than in Sections 6.2 above and 8.3 below where we use R function `jacobian` from R package `numDeriv`. Here (for no particular reason) we use R function `grad` from the same package. R function `jacobian` differentiates vector-to-vector functions. R function `grad` differentiates vector-to-scalar functions. When both are applied to a vector-to-scalar function (which is OK because R cannot tell the difference between scalars and vectors of length one), `jacobian` produces a matrix with one row and `grad` produces a vector. This means we put the transpose in a different place when calculating `my.cov`. If we had used R function `jacobian` instead, we could have made this code look like our other examples.

7.6.2 Offspring

And offspring

```
change.offspring <- mclapply(moo.offspring, function(x)
    mclapply(x, function(y) {
    grad <- grad(fitness_change, y$estimates)
    my.vcov <- t(grad) %*% y$vcov %*% grad
    return(list(estimates = fitness_change(y$estimates), vcov = my.vcov))
}))
```

7.7 Tables

First we map site keys to site names.

```
site.translate <- c(CS = "CERA", GC = "Grey Cloud Dunes", KW = "McCarthy Lake")
```

Turn the calculations above into tables

```
doit <- function(w, cap, lab) {
    e <- lapply(w, function(x) lapply(x, function(y) y$estimates)) |> unlist()
    se <- lapply(w, function(x) lapply(x, function(y) y$vcov)) |> unlist() |>
        sqrt()
    foo <- cbind(estimates = e, std.err. = se)
    site <- substr(rownames(foo), 1, 2)
    site.cs <- match("CS", site)
    site.gc <- match("GC", site)
    site.kw <- match("KW", site)
    year <- substr(rownames(foo), 4, 7)
    rownames(foo) <- year
    kbl(foo, caption = cap, label = lab,
        format = "latex", escape = FALSE, booktabs = TRUE, digits = 4) |>
        kable_styling() |>
```

Table 1: Intragenerational Change in Mean Fitness Due to Selection on Parental Cohort

	estimates	std.err.
CERA		
2015	0.1527	0.1044
2016	0.4823	0.0977
2017	0.1099	0.0712
Grey Cloud Dunes		
2015	0.1683	0.0148
2016	0.1247	0.0238
2017	0.4924	0.0916
McCarthy Lake		
2015	0.4043	0.0582
2016	0.3486	0.0671
2017	0.2317	0.0292

Table 2: Intragenerational Change in Mean Fitness Due to Selection on Offspring Cohort

	estimates	std.err.
CERA		
2016	0.2764	0.1007
2017	0.1471	0.0488
Grey Cloud Dunes		
2016	2.1993	0.1683
2017	0.1882	0.1608
McCarthy Lake		
2016	2.3200	0.3940
2017	0.6887	0.1580

```
        pack_rows(site.translate["GC"], min(site.gc), max(site.gc),
            indent = FALSE) |>
        pack_rows(site.translate["KW"], min(site.kw), max(site.kw),
            indent = FALSE) |>
        pack_rows(site.translate["CS"], min(site.cs), max(site.cs),
            indent = FALSE)
}
```

The following makes Table 1 below.

```
doit(change.parents,
"Intragenerational Change in Mean Fitness Due to Selection on Parental Cohort",
    "changeParents")
```

The following makes Table 2 below.

```
doit(change.offspring,
"Intragenerational Change in Mean Fitness Due to Selection on Offspring Cohort",
    "changeOffspring")
```

8 Comparison of Mean Fitness in Parents and Offspring

8.1 Re-arrange Data

Now we want to make inter-dataset comparisons and inter-year comparisons. To continue with functional programming we need to temporarily step outside of functional programming and use some plain old for loops to re-assemble the data the way we want it. All of our comparisons in this section will involve "parents" in one year and "parents" in the following year (these are pedigreed individuals, and individuals planted in one year have full sibs planted in the next. The "offspring" are open pollinated individuals (mothers known, fathers unknown) whose mothers are individuals among the "parents" in the previous year.

We collect each of these into one vector with one variance-covariance matrix so we can apply one function to them to get estimates and differentiate that one function numerically to apply the delta method to get standard errors.

```
redata <- list()
for (site in names(moo.offspring)) {
    for (year in names(moo.offspring[[site]])) {
        year.before <- as.character(as.numeric(year) - 1)
        moo.prev <- moo.parents[[site]][[year.before]]
        moo.par <- moo.parents[[site]][[year]]
        stopifnot(names(moo.prev$estimates) == names(moo.par$estimates))
        moo.off <- moo.offspring[[site]][[year]]
        stopifnot(names(moo.off$estimates) %in% names(moo.prev$estimates))
        idx <- match(names(moo.off$estimates), names(moo.prev$estimates))
        my.off.est <- rep(0, length(moo.prev$estimates))
        my.off.vcov <- matrix(0, length(my.off.est), length(my.off.est))
        my.off.est[idx] <- moo.off$estimates
        my.off.vcov[idx, idx] <- moo.off$vcov
        redata[[site]][[year]] <- list( estimates =
            c(moo.prev$estimates, moo.par$estimates, my.off.est),
            vcov = bdiag(moo.prev$vcov, moo.par$vcov, my.off.vcov))
    }
}
```

We say that offspring are in the same "family" as parental individuals (pedigreed crosses, whether parents of these offspring or not) if the grandsire of the offspring (the maternal grandsire since sires of offspring are unknown) is the same as the sire of a parental (pedigreed) individual. The code in the function involving R vector `idx` is there to make the estimates and variance-covariance matrices for offspring match those for "parents". We have estimates for each "family" regardless of whether that "family" occurs in the offspring (was planted in the offspring experiment) or not. Fitness is "estimated" to be zero for those families that do not occur in the data (were not planted).

8.2 Functions of Interest

We calculate four functions of interest.

- total change in mean fitness (parents to offspring)

$$\bar{\mu}_{\text{total}} = \text{mean}\left(\mu_{\text{off}} \cdot \frac{\mu_{\text{prev}}}{\text{mean}(\mu_{\text{prev}})}\right) - \text{mean}(\mu_{\text{prev}}) \tag{7}$$

  offspring family mean fitness weighted by relative fitness of their family in the parental generation (because in the experiment offspring were not planted according to their frequency, so we must correct for that)

Which is divided into

- change due to selection (what Fisher's fundamental theorem of natural selection is supposed to predict) in the parents given by (6) above. In this case

$$\text{mean}\left(\mu_{\text{prev}} \cdot \frac{\mu_{\text{prev}}}{\text{mean}(\mu_{\text{prev}})}\right) - \text{mean}(\mu_{\text{prev}}) \tag{8}$$

- environmental change

$$\bar{\mu}_{\text{environment}} = \text{mean}(\mu_{\text{par}}) - \text{mean}(\mu_{\text{prev}}) \tag{9}$$

  difference between pedigreed individuals (which are full sibs) in different years. (Of course, this includes some genetic change because full sibs are not clones. But it does unbiasedly estimate the change due to different environments in the two years because the individuals are random samples from their pedigreed families.)

- residual change, the part of the total change not due to environment or selection

$$\bar{\mu}_{\text{residual}} = \bar{\mu}_{\text{total}} - \bar{\mu}_{\text{selection}} - \bar{\mu}_{\text{environment}} \tag{10}$$

  This may involve recombination, gene-environment interaction, gene-gene interaction or anything else not accounted for in additive effects of selection and environment.

8.3 R Function

Here is our function to compute estimates of interest.

```
foo <- function(theta) {
    stopifnot(is.atomic(theta))
    stopifnot(is.vector(theta))
    stopifnot(is.numeric(theta))
    stopifnot(is.finite(theta))
    stopifnot(length(theta) %% 3 == 0)
    nmu <- length(theta) %/% 3
    is.prev <- seq_along(theta) <= nmu
    is.off <- seq_along(theta) > 2 * nmu
    is.par <- (! (is.prev | is.off))
    mu.prev <- theta[is.prev]
    mu.par <- theta[is.par]
    mu.off <- theta[is.off]

    delta.total <- sum(mu.off * mu.prev) / sum(mu.prev) - mean(mu.prev)
    delta.environ <- mean(mu.par - mu.prev)
    delta.fftns <- mean((mu.prev / mean(mu.prev) - 1) * mu.prev)
    delta.non.fftns <- delta.total - delta.environ - delta.fftns
    c(total = delta.total, environmental = delta.environ,
        selection = delta.fftns, residual = delta.non.fftns)
}
```

Try it out.

```
e <- redata[[1]][[1]]$estimates
foo(e)
```

```
##         total environmental     selection      residual
##     0.3911664     2.8458285     0.1527046    -2.6073667
```

So now we want to use this function to calculate estimates and standard errors.

```
fout <- mclapply(redata, function(x) mclapply(x, function(y) {
    e <- foo(y$estimates)
    jack <- jacobian(foo, y$estimates)
    my.vcov <- jack %*% y$vcov %*% t(jack)
    list(estimates = e, std.err. = sqrt(as.vector(diag(my.vcov))))
}))
```

Our use of the delta method here is just like in Section 6.2 above, where it was explained.

Look at one, just to see we are OK.

```
fout[[1]][[2]]
```

```
## $estimates
##         total environmental     selection      residual
##    -3.2343080    -2.5648810     0.4822954    -1.1517225
##
## $std.err.
## [1] 0.41914484 0.55747044 0.09774297 0.39320758
```

Now make a table for this.

```
doit.too <- function(w) {
    e <- lapply(w, function(x) lapply(x, function(y) y$estimates)) |> unlist()
    se <- lapply(w, function(x) lapply(x, function(y) y$std.err.)) |> unlist()
    foo <- cbind(estimates = e, std.err. = se)
    site <- substr(rownames(foo), 1, 2)
    site.cs <- "CS" == site
    site.gc <- "GC" == site
    site.kw <- "KW" == site
    year <- substr(rownames(foo), 4, 7)
    year.2016 <- "2016" == year
    year.2017 <- "2017" == year
    kind <- substr(rownames(foo), 9, 100)
    rownames(foo) <- kind
    kbl(foo, caption = "Estimates of Change in Mean Fitness", label = "decomp",
        format = "latex", escape = FALSE, booktabs = TRUE, digits = 4) |>
        kable_styling() |>
        pack_rows(paste(site.translate["GC"], "2015-2016"),
            min(which(site.gc & year.2016)),
            max(which(site.gc & year.2016)),
            indent = FALSE) |>
        pack_rows(paste(site.translate["GC"], "2016-2017"),
            min(which(site.gc & year.2017)),
            max(which(site.gc & year.2017)),
            indent = FALSE) |>
        pack_rows(paste(site.translate["KW"], "2015-2016"),
            min(which(site.kw & year.2016)),
            max(which(site.kw & year.2016)),
            indent = FALSE) |>
        pack_rows(paste(site.translate["KW"], "2016-2017"),
            min(which(site.kw & year.2017)),
            max(which(site.kw & year.2017)),
            indent = FALSE) |>
        pack_rows(paste(site.translate["CS"], "2015-2016"),
            min(which(site.cs & year.2016)),
```

Table 3: Estimates of Change in Mean Fitness

	estimates	std.err.
CERA 2015-2016		
total	0.3912	0.4744
environmental	2.8458	0.5991
selection	0.1527	0.1044
residual	-2.6074	0.4694
CERA 2016-2017		
total	-3.2343	0.4191
environmental	-2.5649	0.5575
selection	0.4823	0.0977
residual	-1.1517	0.3932
Grey Cloud Dunes 2015-2016		
total	0.0559	0.1699
environmental	-1.0985	0.1734
selection	0.1683	0.0148
residual	0.9861	0.1334
Grey Cloud Dunes 2016-2017		
total	1.7586	0.3778
environmental	3.4862	0.4422
selection	0.1247	0.0238
residual	-1.8523	0.5657
McCarthy Lake 2015-2016		
total	-0.6403	0.4167
environmental	-1.3173	0.3885
selection	0.4043	0.0582
residual	0.2728	0.3775
McCarthy Lake 2016-2017		
total	-0.4672	0.2946
environmental	-0.5562	0.2746
selection	0.3486	0.0671
residual	-0.2597	0.1982

```
            max(which(site.cs & year.2016)),
            indent = FALSE) |>
        pack_rows(paste(site.translate["CS"], "2016-2017"),
            min(which(site.cs & year.2017)),
            max(which(site.cs & year.2017)),
            indent = FALSE)
}
doit.too(fout)
```

The code chunk above makes Table 3 on this page.

9 Plotting the Decomposition

We are going to try to show these numbers and standard errors in a plot. We will just do one example in this section.

```
site <- "GC"
year <- "2016"
year.before <- as.character(as.numeric(year) - 1)
f <- fout[[site]][[year]]
names(f)
```

```
## [1] "estimates" "std.err."
```

Critical value.

```
conf.level <- 0.95
crit <- qnorm((1 + conf.level) / 2)
```

The horizontal length of the arrows in this plot are entirely meaningless. They have to be different because we don't want the error bars to overlap.

```
cap <- paste0("Estimates in Table 3 with Error Bars.",
   "  Horizontal coordinate is meaningless, but short vectors add up",
   " to long vector.",
   "  Vertical bars are approximate 95% confidence intervals.",
   "  Site is ", site.translate[site], ".",
   "  Total is difference of parents (", year.before, ") and offspring (",
   year, ").",
   "  Selection is difference due to selection in parents (", year.before, ").",
   "  Environmental is difference due to different environments in the",
   " two years: difference of parents (", year.before, ") and full sibs grown",
   " along with offspring (", year, ").",
   "  Residual is Total $-$ Selection $-$ Environmental, mostly genetic change",
   " not due to selection in parents or gene-environment or gene-gene",
   " interaction.")
```

```
par(mar = c(1, 4, 0, 0) + 0.1)
xlength <- c(3, 0.9, 1.0, 1.1)
errbar(x = xlength, y = f$estimates,
   yplus = f$estimates + crit * f$std.err.,
   yminus = f$estimates - crit * f$std.err.,
   axes = FALSE, xlim = c(0, 3),
   ylab = "change in mean fitness",
   pch = NA_integer_)
box()
axis(side = 2)
arrows(x0 = rep(0, 4), x1 = xlength, y0 = rep(0, 4),
   y1 = f$estimates)

# now have to do extraordinarily painful calculation of angles to rotate text
par.pin <- par("pin")
par.plt <- par("plt")
par.usr <- par("usr")
par.plt <- c(diff(par.plt[1:2]), diff(par.plt[3:4]))
par.usr <- c(diff(par.usr[1:2]), diff(par.usr[3:4]))
foo <- par.pin * par.plt / par.usr
foo
```

```
## [1] 1.521159 1.390196
```

```
angles <- atan(f$estimates / xlength * foo[2] / foo[1]) / pi * 180
angles
```

```
##         total environmental     selection      residual
##     0.9756314   -48.1250076     8.7435099    39.3280024
```

```
radians <- 2 * pi * angles / 360
cos(radians)
```

```
##         total environmental     selection      residual
##     0.9998550     0.6675076     0.9883787     0.7735306
```

```
sin(radians)
```

```
##         total environmental     selection      residual
##    0.01702716   -0.74460296    0.15201143    0.63375900
```

```
delta.text <- 0.2
x <- xlength
y <- f$estimates
n <- names(y)
for (i in seq_along(f$estimates))
    text(x = x[i] / 2 - sin(radians[i]) * delta.text / foo[1],
    y = y[i] / 2 + cos(radians[i]) * delta.text / foo[2],
    labels = n[i], col = "darkgray", srt = angles[i])
```

The plot made is Figure 1 below. We might want to say that the arrow labeled "selection" occurs earlier than the other two arrows, but the endpoints of the arrows are not points in time. The arrow labeled "total" is the difference in fitness in the "parents" which takes the entire first year to play out and in the "offspring" which takes the entire second year to play out. Ditto for the arrow labeled "environment". But the arrow labeled "selection" describes what happens only in the "parents" which plays out in the first year. So the arrow labeled "residual" is even more complicated.

Rather than just continue here with the other plots, we move them to a later section.

10 Fitness (Rather than Change Therein)

In this section we revisit change in mean fitness (7) above, and separately calculate the two mean fitnesses that it is the difference of. These are the mean fitness of the parental generation

$$\text{mean}(\mu_{\text{prev}}) \tag{11}$$

and the mean fitness of the offspring generation

$$\text{mean}\left(\mu_{\text{off}} \cdot \frac{\mu_{\text{prev}}}{\text{mean}(\mu_{\text{prev}})}\right) \tag{12}$$

Then we follow the pattern of Section 7 above *mutatis mutandis*.

10.1 Parents

First we apply (11) to all parental experiments (all years, whether they were followed by offspring or not).

```
fitness.parents <- mclapply(moo.parents, function(x) mclapply(x, function(y) {
    e <- y$estimates
    grad <- rep(1 / length(e), length(e)) # this is easy calculus
    my.vcov <- t(grad) %*% y$vcov %*% grad
    return(list(estimates = mean(e), vcov = my.vcov))
}))
```

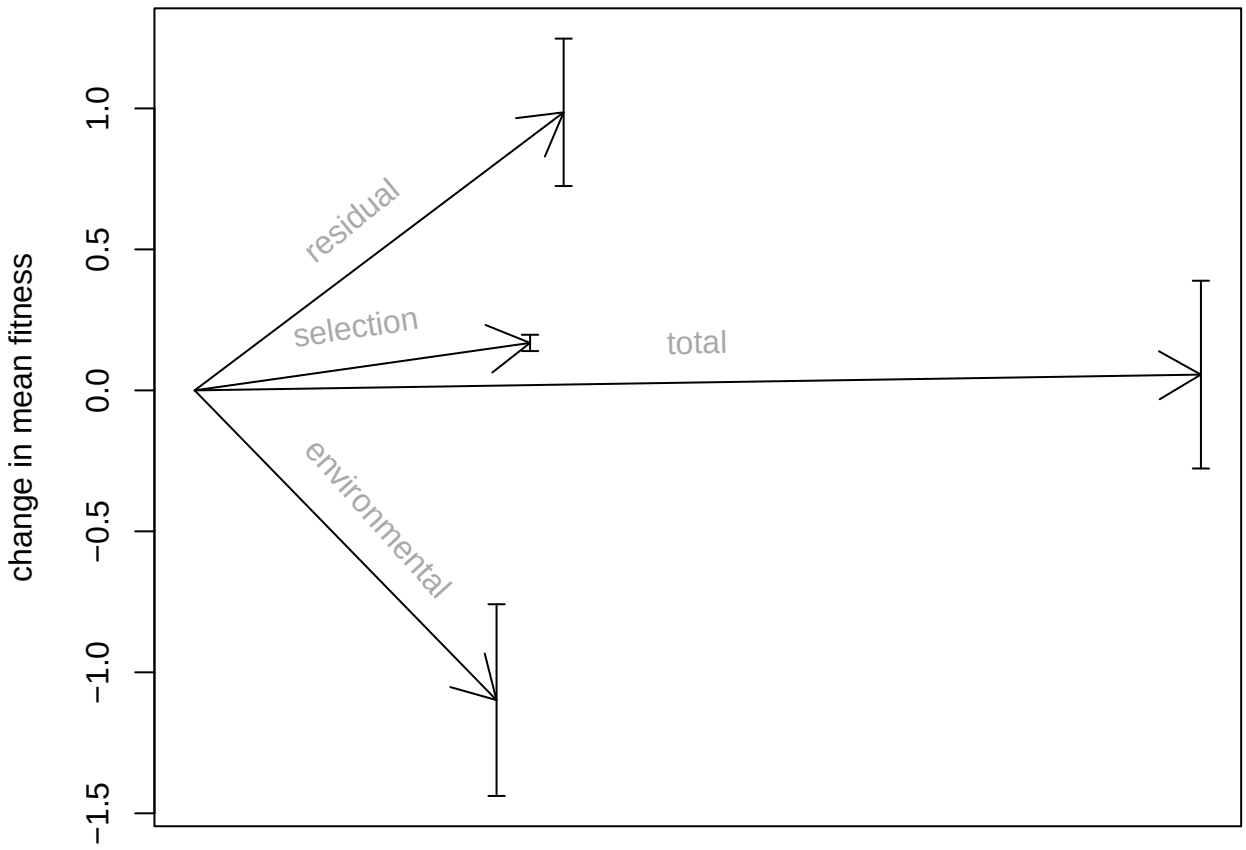

Figure 1: Estimates in Table 3 with Error Bars. Horizontal coordinate is meaningless, but short vectors add up to long vector. Vertical bars are approximate 95% confidence intervals. Site is Grey Cloud Dunes. Total is difference of parents (2015) and offspring (2016). Selection is difference due to selection in parents (2015). Environmental is difference due to different environments in the two years: difference of parents (2015) and full sibs grown along with offspring (2016). Residual is Total − Selection − Environmental, mostly genetic change not due to selection in parents or gene-environment or gene-gene interaction.

Table 4: Mean Fitness of Parental Populations

	estimates	std.err.
CERA		
2015	0.7947	0.4335
2016	3.6406	0.4136
2017	1.0757	0.3738
Grey Cloud Dunes		
2015	2.0216	0.1430
2016	0.9231	0.0982
2017	4.4092	0.4311
McCarthy Lake		
2015	2.7198	0.2959
2016	1.4025	0.2518
2017	0.8463	0.1096

Here we are using R function `grad` in the delta method like we explained in Section 7.6.1 above.

We make a table for this like Table 1. The table we are making is Table 4.

```
doit(fitness.parents, "Mean Fitness of Parental Populations", "fitnessParents")
```

10.2 Offspring

Now we apply (12) to all offspring experiments. Since this also involves parent data, we are following Section 8 *mutatis mutandis* for this calculation.

First we redefine R function `foo` above to calculate offspring fitness

```
foo <- function(theta) {
    stopifnot(is.atomic(theta))
    stopifnot(is.vector(theta))
    stopifnot(is.numeric(theta))
    stopifnot(is.finite(theta))
    stopifnot(length(theta) %% 3 == 0)
    nmu <- length(theta) %/% 3
    is.prev <- seq_along(theta) <= nmu
    is.off <- seq_along(theta) > 2 * nmu
    is.par <- (! (is.prev | is.off))
    mu.prev <- theta[is.prev]
    mu.par <- theta[is.par]
    mu.off <- theta[is.off]

    sum(mu.off * mu.prev) / sum(mu.prev)
}
```

Then we repeat Section 8 above except for using the `foo` defined here rather than the `foo` defined there.

```
fitness.offspring <- mclapply(redata, function(x) mclapply(x, function(y) {
    e <- foo(y$estimates)
    jack <- jacobian(foo, y$estimates)
    my.vcov <- jack %*% y$vcov %*% t(jack)
    list(estimates = e, vcov = as.matrix(my.vcov))
}))
```

Table 5: Mean Fitness of Offspring Populations

	estimates	std.err.
CERA		
2016	1.1859	0.1974
2017	0.4063	0.0746
Grey Cloud Dunes		
2016	2.0775	0.0895
2017	2.6817	0.3661
McCarthy Lake		
2016	2.0796	0.2786
2017	0.9353	0.1508

(We changed the second component of the result from `std.err.` to `vcov` in order to match the parental calculation. The reason for the `as.matrix` is to convert a sparse matrix from R package `Matrix` into an ordinary (core R) matrix.)

But our table is done by the same code as for the table for parents in this section. The table we are making is Table 5.

```
doit(fitness.offspring, "Mean Fitness of Offspring Populations",
    "fitnessOffspring")
```

11 More Plots

We ought to use a loop here but R markdown does not seem to be up to the task. So we use five more code chunks, which we hide.

12 Write Out Stuff For Paper

12.1 To Do

Table 1 for paper is Table 4 above. Except we also want post-selection mean fitness too.

Table 2 for paper is Table 5 above.

Table 3 for paper is Table 3 above. Except FFTNS prediction should be included too. And For that we need to redo the calculation from Geyer *et al.* (2022) with correct standard errors (no infinite standard errors because we found the cause of that (Section 3.3 above).

Figure 1, six panels, error bars (or something similar) for selection, environment, residual, total (in that order left-to-right). No arrows.

12.2 New Table 1

For this we need R object `fitness.parents` to have the numbers in 4. and the numbers that are the first term in ((8))

$$\text{mean}\left(\mu \cdot \frac{\mu}{\text{mean}(\mu)}\right)$$

```
moo.after <- function(mu) mean(mu * mu / mean(mu))
fitness.parents.after <- mclapply(moo.parents, function(x)
    mclapply(x, function(y) {
```

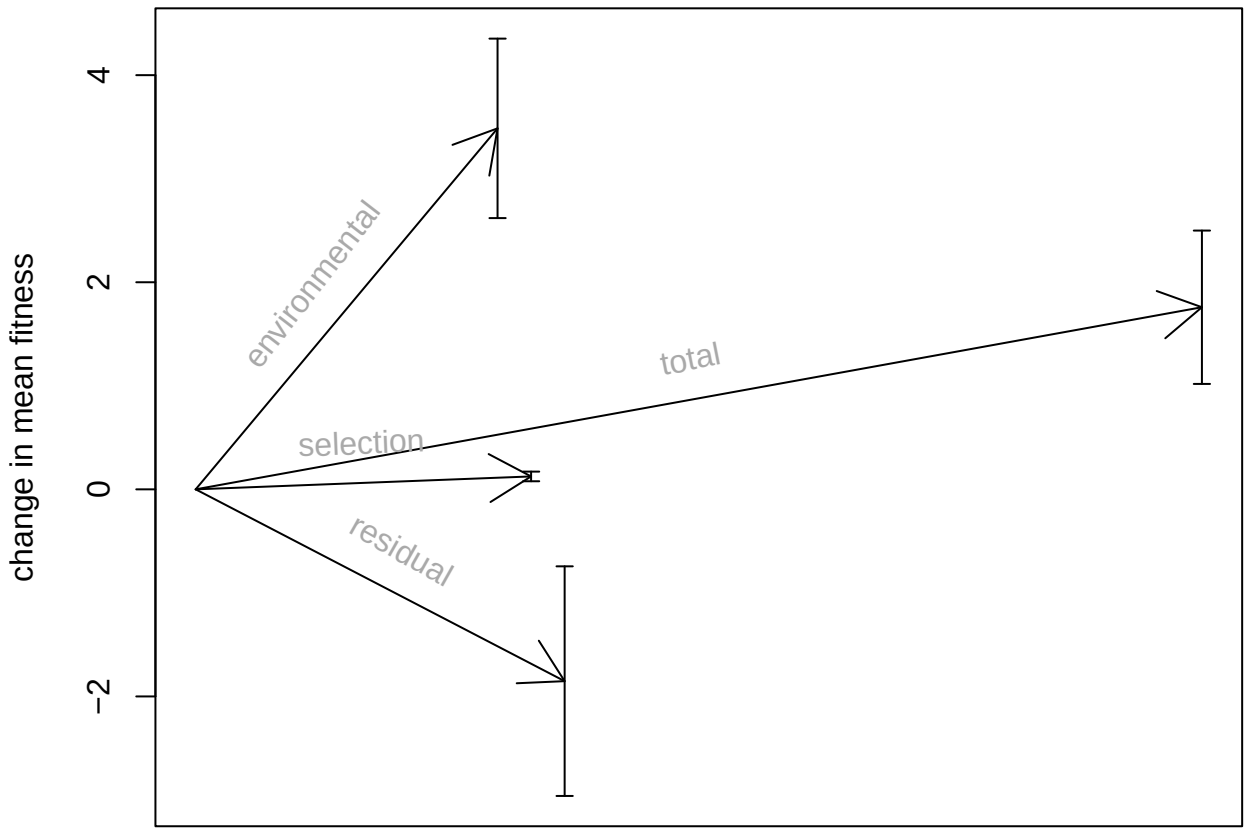

Figure 2: Estimates in Table 3 with Error Bars. Horizontal coordinate is meaningless, but short vectors add up to long vector. Vertical bars are approximate 95% confidence intervals. Site is Grey Cloud Dunes. Total is difference of parents (2016) and offspring (2017). Selection is difference due to selection in parents (2016). Environmental is difference due to different environments in the two years: difference of parents (2016) and full sibs grown along with offspring (2017). Residual is Total − Selection − Environmental, mostly genetic change not due to selection in parents or gene-environment or gene-gene interaction.

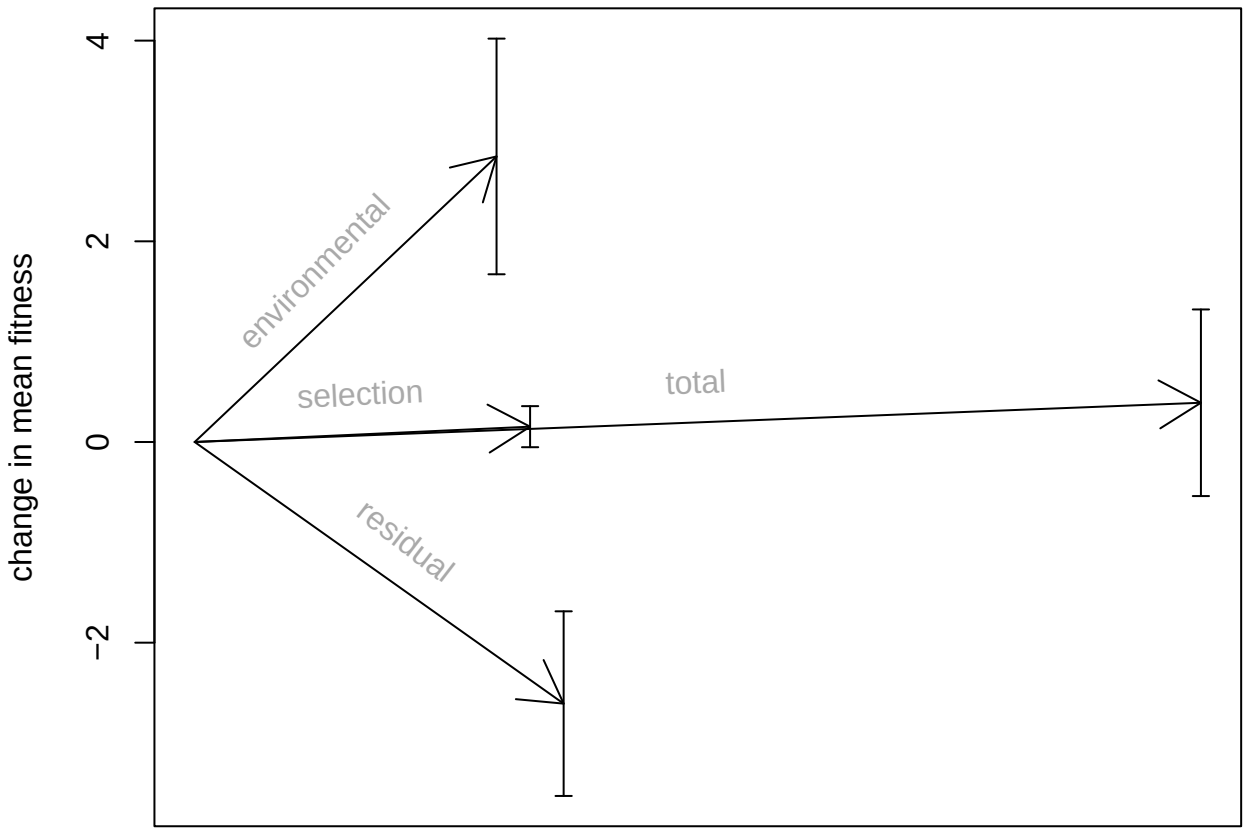

Figure 3: Estimates in Table 3 with Error Bars. Horizontal coordinate is meaningless, but short vectors add up to long vector. Vertical bars are approximate 95% confidence intervals. Site is CERA. Total is difference of parents (2015) and offspring (2016). Selection is difference due to selection in parents (2015). Environmental is difference due to different environments in the two years: difference of parents (2015) and full sibs grown along with offspring (2016). Residual is Total − Selection − Environmental, mostly genetic change not due to selection in parents or gene-environment or gene-gene interaction.

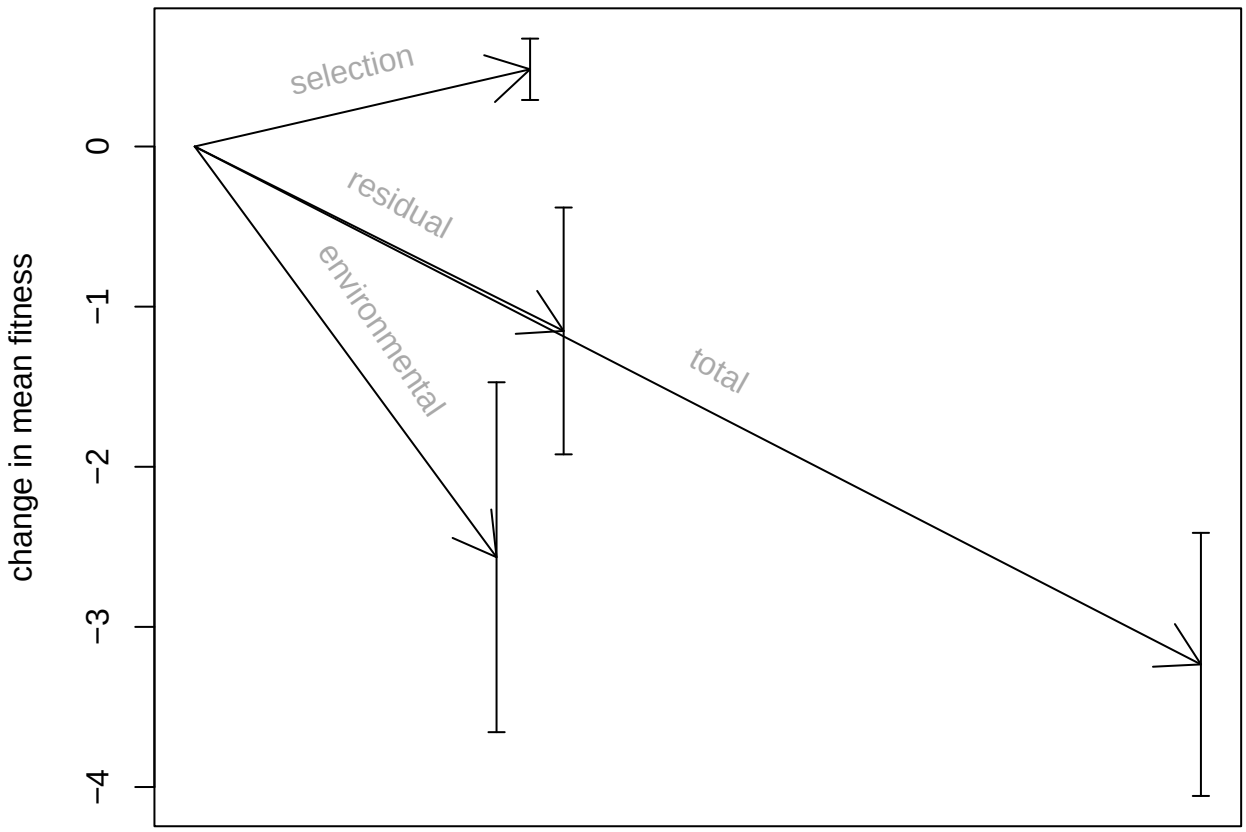

Figure 4: Estimates in Table 3 with Error Bars. Horizontal coordinate is meaningless, but short vectors add up to long vector. Vertical bars are approximate 95% confidence intervals. Site is CERA. Total is difference of parents (2016) and offspring (2017). Selection is difference due to selection in parents (2016). Environmental is difference due to different environments in the two years: difference of parents (2016) and full sibs grown along with offspring (2017). Residual is Total − Selection − Environmental, mostly genetic change not due to selection in parents or gene-environment or gene-gene interaction.

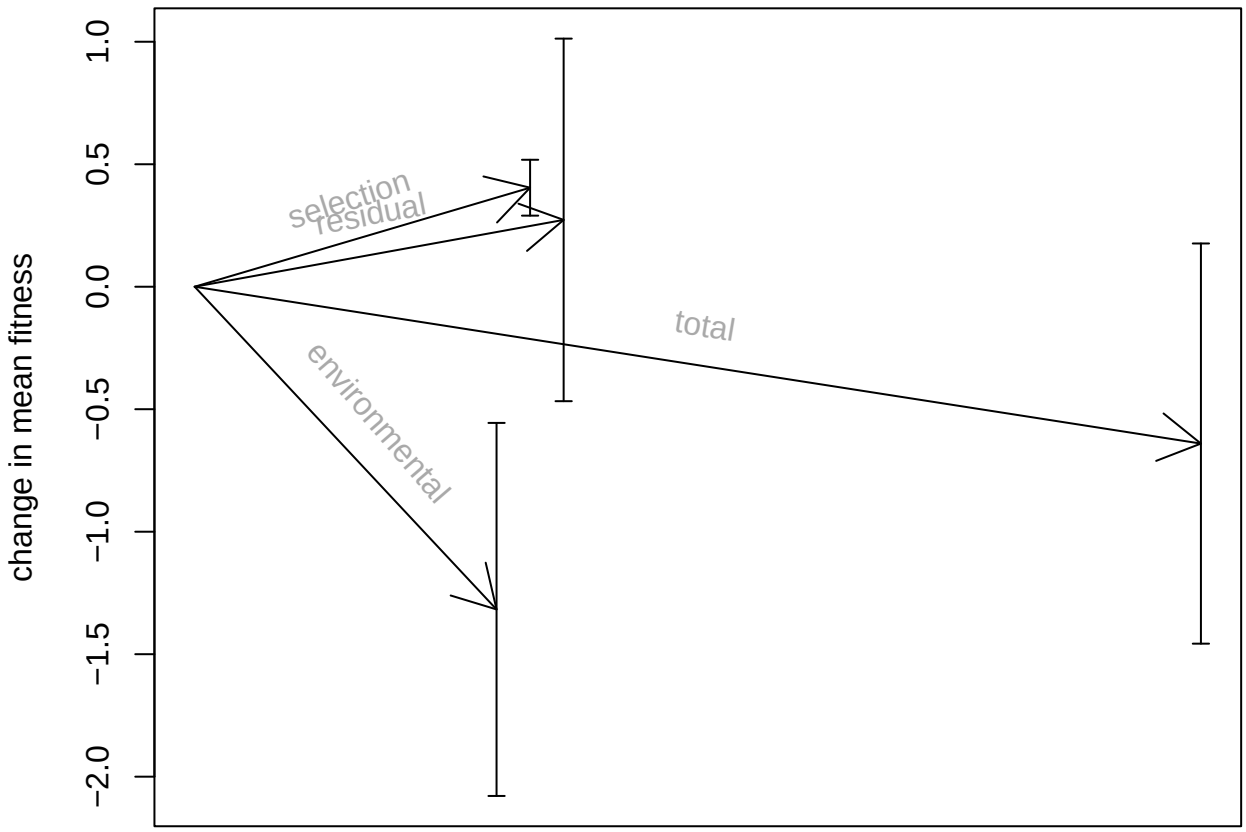

Figure 5: Estimates in Table 3 with Error Bars. Horizontal coordinate is meaningless, but short vectors add up to long vector. Vertical bars are approximate 95% confidence intervals. Site is McCarthy Lake. Total is difference of parents (2015) and offspring (2016). Selection is difference due to selection in parents (2015). Environmental is difference due to different environments in the two years: difference of parents (2015) and full sibs grown along with offspring (2016). Residual is Total − Selection − Environmental, mostly genetic change not due to selection in parents or gene-environment or gene-gene interaction.

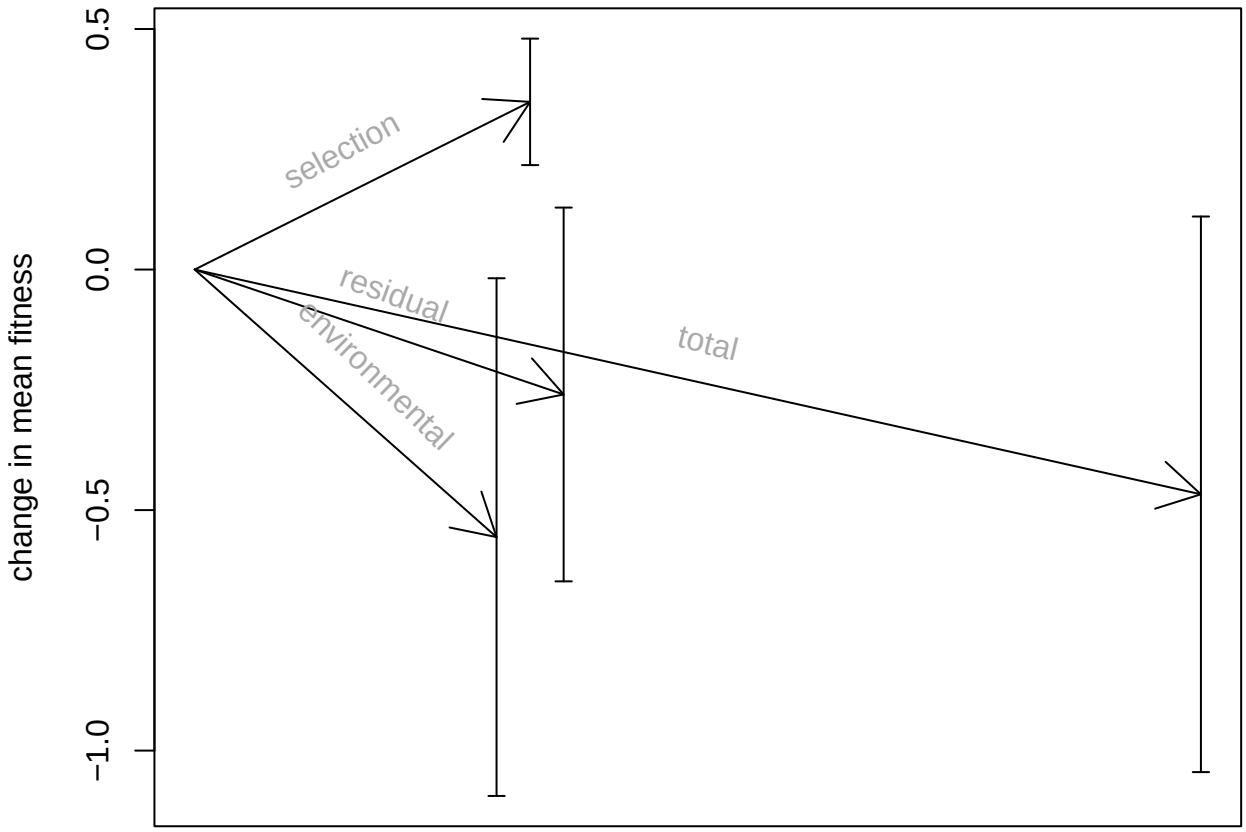

Figure 6: Estimates in Table 3 with Error Bars. Horizontal coordinate is meaningless, but short vectors add up to long vector. Vertical bars are approximate 95% confidence intervals. Site is McCarthy Lake. Total is difference of parents (2016) and offspring (2017). Selection is difference due to selection in parents (2016). Environmental is difference due to different environments in the two years: difference of parents (2016) and full sibs grown along with offspring (2017). Residual is Total − Selection − Environmental, mostly genetic change not due to selection in parents or gene-environment or gene-gene interaction.

```
    e <- moo.after(y$estimates)
    g <- grad(moo.after, y$estimates)
    my.vcov <- drop(t(g) %*% y$vcov %*% g)
    return(list(estimates = mean(e), vcov = my.vcov))
}))

doit.helper <- function(w, cap, lab) {
    e <- lapply(w, function(x) lapply(x, function(y) y$estimates)) |> unlist()
    se <- lapply(w, function(x) lapply(x, function(y) y$vcov)) |> unlist() |>
        sqrt()
    cbind(estimates = e, std.err. = se)
}
foo <- cbind(doit.helper(fitness.parents), doit.helper(fitness.parents.after))
colnames(foo) <- c("e1", "s1", "e2", "s2")
```

Continue with code from R function `doit`. We have to stop using R package `kableExtra` because it does not work for Microsoft Word, which this document is not, but we eventually want to get to. So we switch to R package `flextable`.

```
set_flextable_defaults(fonts_ignore=TRUE)
foot <- data.frame(foo) |>
    (\(x) transform(x, site = substr(rownames(x), 1, 2)))() |>
    (\(x) transform(x, year = substr(rownames(x), 4, 7)))() |>
    transform(site = site.translate[site]) |>
    transform(site = as.factor(site)) |>
    as.data.table() |>
    as_grouped_data(groups = c("site"),
        columns = c("year", "e1", "s1", "e2", "s2")) |>
    flextable() |>
    set_header_labels(year = "Year", e1 = "Estimate", s1 = "Std. Error",
        e2 = "Estimate", s2 = "Std. Error") |>
    colformat_double(digits = 4) |>
    add_header_row(colwidths = c(1, 1, 2, 2),
        values = c("", "", "before selection", "after selection"))
foot
```

Table 6: Mean Fitness of Parental Populations

		before selection		after selection	
site	Year	Estimate	Std. Error	Estimate	Std. Error
CERA					
	2015	0.7947	0.4335	0.9474	0.5366
	2016	3.6406	0.4136	4.1229	0.4997
	2017	1.0757	0.3738	1.1856	0.4423
Grey Cloud Dunes					
	2015	2.0216	0.1430	2.1899	0.1538
	2016	0.9231	0.0982	1.0478	0.1187
	2017	4.4092	0.4311	4.9016	0.5130

Table 6: Mean Fitness of Parental Populations

		before selection		after selection	
site	Year	Estimate	Std. Error	Estimate	Std. Error
McCarthy Lake					
	2015	2.7198	0.2959	3.1241	0.2825
	2016	1.4025	0.2518	1.7511	0.3162
	2017	0.8463	0.1096	1.0780	0.1341

OK. Ship out to Microsoft Word.

```
foot <- set_caption(foot, "Mean Fitness of Parental Populations")
save_as_docx(foot, path = "table1.docx")
```

What a struggle! And we still have an inexplicable error (which does not show in the document, only as a compilation warning).

12.3 New Table 2

Just like new table 1 except nothing extra to calculate.

```
foot <- doit.helper(fitness.offspring) |>
    as.data.frame() |>
    (\(x) transform(x, site = substr(rownames(x), 1, 2)))() |>
    (\(x) transform(x, year = substr(rownames(x), 4, 7)))() |>
    transform(site = site.translate[site]) |>
    transform(site = as.factor(site)) |>
    as.data.table() |>
    as_grouped_data(groups = c("site"),
        columns = c("year", "estimates", "std.err.")) |>
    flextable() |>
    set_header_labels(year = "Year", estimates = "Estimate",
        std.err. = "Std. Error") |>
    colformat_double(digits = 4)
foot
```

Table 7: Mean Fitness of Offspring Populations

site	Year	Estimate	Std. Error
CERA			
	2016	1.1859	0.1974
	2017	0.4063	0.0746
Grey Cloud Dunes			
	2016	2.0775	0.0895
	2017	2.6817	0.3661

Table 7: Mean Fitness of Offspring Populations

site	Year	Estimate	Std. Error
McCarthy Lake			
	2016	2.0796	0.2786
	2017	0.9353	0.1508

OK. Ship out to Microsoft Word.

```
foot <- set_caption(foot, "Mean Fitness of Offspring Populations")
save_as_docx(foot, path = "table2.docx")
```

12.4 Recalculate Results from Evolution Correction

12.4.1 Differences Between this Paper Previous Ones

We are going to redo some of the calculations of Geyer *et al.* (2022) in order to compare them with this paper. The approaches are rather different. Neither is wrong (except in one detail mentioned presently). They are just addressing different (albeit related) things.

In particular, Geyer *et al.* (2022) calculates the prediction from Fisher's fundamental theorem of natural selection (FFTNS), which is additive genetic variance for fitness divided by mean fitness. In this paper, we do not (at least up to now) calculate any such thing. Instead we calculate what this quantity (according to FFTNS) predicts, actual observed change in mean fitness. This does not involve any variance (additive genetic or otherwise).

The error in Geyer *et al.* (2022) is that some of these quantities had infinite standard errors, which is a mistake. That is due solely to not recognizing (Section 3.3 above) that some site-year combinations have subsampling and some do not have it, so we need to do different analyses (Section 4 above) for the different cases.

12.4.2 What Is To Be Done

Geyer *et al.* (2022), Section 12.3.2 (following Geyer and Shaw (2013)) define an R function `map.too` that takes one vector argument `balpha` that decomposes into two parts,

- a scalar `b` which is the additive genetic effect for a single hypothetical individual (not necessarily one in the observed data) and
- a vector `alpha` which is the fixed effects part of the random effects aster model.

To turn this from computer code to math, we let the R function `map.too` correspond to a mathematical function f of two arguments, so $f(b, \alpha)$ is the value of the function. Then our FFTNS prediction is

$$Q(\alpha, \nu) = \left(\frac{\partial f(0, \alpha)}{\partial b} \right)^2 \cdot 4\nu \cdot \frac{1}{f(0, \alpha)} \tag{13}$$

where ν is the variance on the canonical parameter scale for parental random effects.

We estimate this, of course, by plugging in MLE $Q(\hat{\alpha}, \hat{\nu})$.

To calculate standard errors we need derivatives, which are given by Geyer *et al.* (2022), Section 12.3.2

$$\frac{\partial Q(\alpha, \nu)}{\partial \alpha} = -\left(\frac{\partial f(0, \alpha)}{\partial b}\right)^2 \cdot 4\nu \cdot \frac{1}{f(0, \alpha)^2} \cdot \frac{\partial f(0, \alpha)}{\partial \alpha} + \frac{\partial f(0, \alpha)}{\partial b} \cdot 8\nu \cdot \frac{1}{f(0, \alpha)} \cdot \frac{\partial^2 f(0, \alpha)}{\partial b \partial \alpha} \tag{14}$$

$$\frac{\partial Q(\alpha, \nu)}{\partial \nu} = \left(\frac{\partial f(0, \alpha)}{\partial b}\right)^2 \cdot 4 \cdot \frac{1}{f(0, \alpha)} \tag{15}$$

12.4.3 Another Function Factory

Rather than exactly reproduce what Geyer *et al.* (2022) do, we combine our approach in Section 6.1 above to make a function factory that produces a function that calculates the mathematical function f described in the preceding section.

As in Section 6.1 above, we develop our function a little bit at a time and then assemble all the bits into the whole function. And. as in that section, we need an object to be the argument of that function to exercise our code bits on.

```
names(rout.parents)
```

```
## [1] "CS" "GC" "KW"
```

```
rout <- rout.parents[[1]][[1]]
rout <- rout.parents$GC[[1]]
```

Now we already have a lot of setup we can just repeat from Section 6.1 above.

```
    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)
```

But we also need a bit more setup, following R function `map.factory.too` in Section 12.3.2 of Geyer *et al.* (2022).

```
    alpha <- rout$alpha
    ifit <- which(names(alpha) == "fit")
    if (length(ifit) != 1)
        stop("no fixed effect named fit")
```

Now we can just copy the function returned by the map factory from Section 12.3.2 of Geyer *et al.* (2022).

Again we write the code in the body of the function before the function itself, so we need an R object to serve as the function argument.

```
balpha <- c(0, alpha)
```

Then the function body is

```
        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]
```

```
## [1] 1.871795
```

Everything seems to be all right, so we put it all together.

```
map.factory.other <- 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)
    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]
    }
}
```

Try it out.

```
map.too <- map.factory.other(rout)
map.too(balpha)
```

```
## [1] 1.871795
```

12.4.4 FFTNS Estimates

We use the function factory defined in the preceding section to make FFTNS estimates.

```
doit.fftns <- function(rout) {
    map.too <- map.factory.other(rout)
    balpha.hat <- c(0, rout$alpha)
    g <- grad(map.too, balpha.hat)
    dmu.db <- g[1]
    mu.hat <- map.too(balpha.hat)
```

```
    nu.hat <- rout$nu["parental"]
    as.numeric(dmu.db^2 * 4 * nu.hat / mu.hat)
}
```

Not too hard. If it isn't clear what these computations are, compare with Section 12.3.2 of Geyer *et al.* (2022).

Try it.

```
doit.fftns(rout)
```

```
## [1] 2.448443
```

Do all.

```
lapply(rout.parents, function(x) lapply(x, doit.fftns))
```

```
## $CS
## $CS$`2015`
## [1] 2.125778
##
## $CS$`2016`
## [1] 2.697465
##
## $CS$`2017`
## [1] 0.9854696
##
##
## $GC
## $GC$`2015`
## [1] 2.448443
##
## $GC$`2016`
## [1] 0.8826717
##
## $GC$`2017`
## [1] 3.899511
##
##
## $KW
## $KW$`2015`
## [1] 4.620264
##
## $KW$`2016`
## [1] 3.347372
##
## $KW$`2017`
## [1] 2.060365
```

Seems to agree with Table 1 of Geyer *et al.* (2022).

12.4.5 FFTNS Estimates with Standard Errors

Again following Geyer *et al.* (2022), Section 12.3.2, we modify R function `doit.fftns` to also compute standard errors.

```
doit.fftns <- function(rout) {
    map.too <- map.factory.other(rout)
```

```
    balpha.hat <- c(0, rout$alpha)
    g <- grad(map.too, balpha.hat)
    h <- hessian(map.too, balpha.hat)
    dmu.db <- g[1]
    dmu.dalpha <- g[-1]
    d2mu.db.dalpha <- h[1, -1]
    mu.hat <- map.too(balpha.hat)
    nu.hat <- rout$nu["parental"]
    dfftns <- c(- 4 * nu.hat * dmu.dalpha * dmu.db^2 / mu.hat^2 +
        8 * nu.hat * dmu.db * d2mu.db.dalpha / mu.hat,
        4 * dmu.db^2 / mu.hat, 0)
    fishinv <- vcov(rout, standard.deviation = FALSE)
    fftns.se <- t(dfftns) %*% fishinv %*% dfftns
    fftns.se <- sqrt(as.vector(fftns.se))
    point.estimate <- dmu.db^2 * 4 * nu.hat / mu.hat
    c(estimate = point.estimate, std.err. = fftns.se)
}
```

Try it.

```
doit.fftns(rout)
```

```
## estimate.parental          std.err.
##         2.4484430         0.8062094
```

Do all.

```
fftns.results <- lapply(rout.parents, function(x) lapply(x, doit.fftns))
fftns.results
```

```
## $CS
## $CS$`2015`
## estimate.parental          std.err.
##          2.125778          2.240112
##
## $CS$`2016`
## estimate.parental          std.err.
##          2.697465          1.209791
##
## $CS$`2017`
## estimate.parental          std.err.
##         0.9854696         0.7613122
##
##
## $GC
## $GC$`2015`
## estimate.parental          std.err.
##         2.4484430         0.8062094
##
## $GC$`2016`
## estimate.parental          std.err.
##         0.8826717         0.3254288
##
## $GC$`2017`
## estimate.parental          std.err.
##          3.899511          4.595502
```

```
##
##
## $KW
## $KW$`2015`
## estimate.parental          std.err.
##      4.620264e+00      9.757819e+06
##
## $KW$`2016`
## estimate.parental          std.err.
##          3.347372          4.743035
##
## $KW$`2017`
## estimate.parental          std.err.
##          2.060365          2.034608
```

Seems to agree with Table 1 of Geyer *et al.* (2022) except for the ones that were `NA` there (which were incorrect).

12.4.6 Wait, What?

But there still seems to be an issue with KW in 2015, which is ridiculously large.

Check that (debug).

```
rout <- rout.parents$KW[["2015"]]
foo <- vcov(rout, standard.deviation = FALSE)
foo
```

```
##                  (Intercept)           fit      varbGerm varbtotal.pods
## (Intercept)     1.651540e-01 -1.652209e-01 -2.021912e-01  -1.749105e-01
## fit            -1.652209e-01  1.653799e-01  2.022581e-01   1.741565e-01
## varbGerm       -2.021912e-01  2.022581e-01  2.765671e-01   2.119477e-01
## varbtotal.pods -1.749105e-01  1.741565e-01  2.119477e-01   1.930110e-01
## parental        5.998696e-07 -6.111614e-07 -5.998696e-07  -5.951429e-07
## block           0.000000e+00  0.000000e+00  0.000000e+00   0.000000e+00
##                     parental block
## (Intercept)     5.998696e-07     0
## fit            -6.111614e-07     0
## varbGerm       -5.998696e-07     0
## varbtotal.pods -5.951429e-07     0
## parental        1.122012e-10     0
## block           0.000000e+00     0
## attr(,"is.alpha")
## [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## attr(,"is.nu")
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
```

```
max(foo)
```

```
## [1] 0.2765671
```

Nothing there. Continue.

```
map.too <- map.factory.other(rout)
balpha.hat <- c(0, rout$alpha)
g <- grad(map.too, balpha.hat)
h <- hessian(map.too, balpha.hat)
dmu.db <- g[1] |> print()
```

```
## [1] 287.0032
```

```
dmu.dalpha <- g[-1] |> print()
```

```
## [1] 323.090218 287.003187   2.464133  31.138701
```

```
d2mu.db.dalpha <- h[1, -1] |> print()
```

```
## [1] -4.689653e+06 -7.499382e+08  2.715189e+02  3.696148e+03
```

```
mu.hat <- map.too(balpha.hat) |> print()
```

```
## [1] 2.545839
```

```
nu.hat <- rout$nu["parental"] |> print()
```

```
##     parental
## 3.569967e-05
```

It seems the results of R function `hessian` from R package `numDeriv` are ridiculous. We have observed similar problems before (not published).

12.4.7 A Likelihood Interval

Hence we produce a likelihood interval, or more precisely an objective-function-based interval, where the objective function in question is the approximation to the log likelihood used by R function `reaster`.

This uses R function `objfun.factory` which is new in R package `aster` in version 1.3-4 which this document requires.

```
objfun <- with(rout, objfun.factory(fixed, random, response,
    obj$pred, obj$fam, as.vector(obj$root), zwz))
theta.hat <- with(rout, c(alpha, b, nu))
objfun(theta.hat)$value
```

```
## [1] -14810.87
```

This approximates minus log likelihood, so the likelihood ratio test statistic is approximated by

```
lrt.factory <- function(objfun, theta.hat) {
    lrt.min <- objfun(theta.hat)$value
    function(theta) 2 * (objfun(theta)$value - lrt.min)
}
lrt <- lrt.factory(objfun, theta.hat)
```

We form a 68.27% confidence interval by maximizing and minimizing a parameter of interest, in this case the FFTNS prediction calculated by `map.too` over the region of the parameter space where R function `lrt` has the value less than or equal to one. We choose this confidence interval to be equivalent to a Wald interval one standard error to either side of the point estimate. Of course, a likelihood interval will not be symmetric about the point estimate. (We will deal with that later.)

Now we make the objective function. This is just the parameter estimate evaluated by R function `map.too` except we change the value to `Inf` ($+\infty$) when we are minimizing and `-Inf` ($-\infty$) when maximizing when we are off the constraint set, that is when R function `lrt` evaluates to greater than 1.

```
objfun.fftns.factory <- function(rout, direction = c("minimize", "maximize")) {
    direction <- match.arg(direction)
    stopifnot(inherits(rout, "reaster"))
    map.too <- map.factory.other(rout)
    objfun <- with(rout, objfun.factory(fixed, random, response,
        obj$pred, obj$fam, as.vector(obj$root), zwz))
```

```
    theta.hat <- with(rout, c(alpha, b, nu))
    lrt.min <- objfun(theta.hat)$value
    lrt <- function(theta) 2 * (objfun(theta)$value - lrt.min)
    is.alpha <- seq_along(theta.hat) <= length(rout$alpha)
    is.nu <- c(rep(FALSE, length(rout$alpha) + length(rout$b)),
        names(rout$nu) == "parental")
    stopifnot(sum(is.nu) == 1)
    function(theta) {
        stopifnot(is.numeric(theta))
        stopifnot(is.finite(theta))
    stopifnot(length(theta) == length(theta.hat))
        if (lrt(theta) > 1) return(if (direction == "minimize") Inf else -Inf)
    alpha <- theta[is.alpha]
    balpha <- c(0, alpha)
    nu <- theta[is.nu]
        g <- grad(map.too, balpha)
        dmu.db <- g[1]
        mu.hat <- map.too(balpha)
        as.numeric(dmu.db^2 * 4 * nu / mu.hat)
    }
}
```

OK. Minimize.

```
oout.dn <- optim(theta.hat, objfun.fftns.factory(rout), method = "SANN",
    control = list(trace = TRUE, parscale = rep(1e-6, length(theta.hat))))
```

```
## sann objective function values
## initial       value 4.620264
## iter     1000 value 4.415412
## iter     2000 value 4.125695
## iter     3000 value 4.087972
## iter     4000 value 4.087972
## iter     5000 value 4.087972
## iter     6000 value 4.087972
## iter     7000 value 4.087972
## iter     8000 value 4.087972
## iter     9000 value 4.087972
## iter     9999 value 3.907399
## final         value 3.907399
## sann stopped after 9999 iterations
```

OK. Maximize.

```
oout.up <- optim(theta.hat, objfun.fftns.factory(rout), method = "SANN",
    control = list(trace = TRUE, parscale = rep(1e-6, length(theta.hat)),
        fnscale = -1))
```

```
## sann objective function values
## initial       value -4.620264
## iter     1000 value -5.182684
## iter     2000 value -5.226548
## iter     3000 value -5.242548
## iter     4000 value -5.242548
## iter     5000 value -5.242548
## iter     6000 value -5.242548
```

```
## iter     7000 value -5.242548
## iter     8000 value -5.242548
## iter     9000 value -5.242548
## iter     9999 value -5.242548
## final         value -5.242548
## sann stopped after 9999 iterations
```

Get our interval.

```
c(oout.dn$value, oout.up$value)
```

```
## [1] 3.907399 5.242548
```

If we redo, do we get a wider interval?

```
oout.dn <- optim(oout.dn$par, objfun.fftns.factory(rout), method = "SANN",
    control = list(trace = TRUE, parscale = rep(1e-6, length(theta.hat))))
```

```
## sann objective function values
## initial       value 3.907399
## iter     1000 value 3.907399
## iter     2000 value 3.907399
## iter     3000 value 3.907399
## iter     4000 value 3.907399
## iter     5000 value 3.907399
## iter     6000 value 3.907399
## iter     7000 value 3.907399
## iter     8000 value 3.907399
## iter     9000 value 3.907399
## iter     9999 value 3.907399
## final         value 3.907399
## sann stopped after 9999 iterations
```

```
oout.up <- optim(oout.up$par, objfun.fftns.factory(rout), method = "SANN",
    control = list(trace = TRUE, parscale = rep(1e-6, length(theta.hat)),
        fnscale = -1))
```

```
## sann objective function values
## initial       value -5.242548
## iter     1000 value -5.439999
## iter     2000 value -5.439999
## iter     3000 value -5.439999
## iter     4000 value -5.439999
## iter     5000 value -5.439999
## iter     6000 value -5.439999
## iter     7000 value -5.439999
## iter     8000 value -5.439999
## iter     9000 value -5.439999
## iter     9999 value -5.439999
## final         value -5.439999
## sann stopped after 9999 iterations
```

```
c(oout.dn$value, oout.up$value)
```

```
## [1] 3.907399 5.439999
```

Hardly changed. So that's it.

Now that we have a method for making (approximate) likelihood-based confidence intervals, perhaps we

should use that instead of standard errors and asymptotic normality. But we won't change horses in the middle of the stream. So we are done. Fix up (by hand) the standard error that is ridiculous.

```
fftns.results$KW[["2015"]]
```

```
## estimate.parental          std.err.
##      4.620264e+00      9.757819e+06
```

```
fftns.results$KW[["2015"]]["std.err."] <-
    diff(c(oout.dn$value, oout.up$value)) / 2
fftns.results$KW[["2015"]]
```

```
## estimate.parental          std.err.
##          4.620264          0.766300
```

12.5 New Table 3

Now we combine Table 3 above, whose numbers are in R object `fout`, with the numbers for FFTNS predictions that were computed in the preceding section and originally were (mostly) in Geyer *et al.* (2022).

```
head(fout)
```

```
## $CS
## $CS$`2016`
## $CS$`2016`$estimates
##         total environmental     selection      residual
##     0.3911664     2.8458285     0.1527046    -2.6073667
##
## $CS$`2016`$std.err.
## [1] 0.4744324 0.5991034 0.1044051 0.4694478
##
##
## $CS$`2017`
## $CS$`2017`$estimates
##         total environmental     selection      residual
##    -3.2343080    -2.5648810     0.4822954    -1.1517225
##
## $CS$`2017`$std.err.
## [1] 0.41914484 0.55747044 0.09774297 0.39320758
##
##
##
## $GC
## $GC$`2016`
## $GC$`2016`$estimates
##         total environmental     selection      residual
##    0.05590167   -1.09852371    0.16828726    0.98613811
##
## $GC$`2016`$std.err.
## [1] 0.16989358 0.17344767 0.01475605 0.13336575
##
##
## $GC$`2017`
## $GC$`2017`$estimates
##         total environmental     selection      residual
##     1.7585895     3.4861619     0.1247013    -1.8522736
##
```

```
## $GC$`2017`$std.err.
## [1] 0.37777618 0.44216853 0.02378542 0.56570753
##
##
##
## $KW
## $KW$`2016`
## $KW$`2016`$estimates
##         total environmental     selection      residual
##    -0.6402514    -1.3173033     0.4042522     0.2727997
##
## $KW$`2016`$std.err.
## [1] 0.41672146 0.38846652 0.05817232 0.37754041
##
##
## $KW$`2017`
## $KW$`2017`$estimates
##         total environmental     selection      residual
##    -0.4672158    -0.5561569     0.3486017    -0.2596607
##
## $KW$`2017`$std.err.
## [1] 0.29462996 0.27455459 0.06707577 0.19820509
```

```
head(fftns.results)
```

```
## $CS
## $CS$`2015`
## estimate.parental          std.err.
##          2.125778          2.240112
##
## $CS$`2016`
## estimate.parental          std.err.
##          2.697465          1.209791
##
## $CS$`2017`
## estimate.parental          std.err.
##         0.9854696         0.7613122
##
##
## $GC
## $GC$`2015`
## estimate.parental          std.err.
##         2.4484430         0.8062094
##
## $GC$`2016`
## estimate.parental          std.err.
##         0.8826717         0.3254288
##
## $GC$`2017`
## estimate.parental          std.err.
##          3.899511          4.595502
##
##
## $KW
## $KW$`2015`
```

```
## estimate.parental          std.err.
##          4.620264          0.766300
##
## $KW$`2016`
## estimate.parental          std.err.
##          3.347372          4.743035
##
## $KW$`2017`
## estimate.parental          std.err.
##          2.060365          2.034608
```

First merge these two lists of lists.

```
qux <- character(0)
quux <- character(0)
quuux <- double(0)
quuuux <- double(0)
for (site in names(fout)) {
    for (year in names(fout[[1]])) {
    prev.year <- as.character(as.numeric(year) - 1)
        lab <- paste0(site.translate[site], " ", prev.year, "-", year)
        foo <- fout[[site]][[year]]
        bar <- fftns.results[[site]][[year]]
    baz <- c(foo$estimates[-4], bar[1], foo$estimates[4])
    baze <- c(foo$std.err.[-4], bar[2], foo$std.err.[4])
    qux <- c(qux, rep(lab, 5))
    quux <- c(quux,
            c("total", "environmental", "selection", "fftns", "residual"))
    quuux <- c(quuux, baz)
    quuuux <- c(quuuux, baze)
    }
}
foo <- data.frame(Subset = qux, change = quux, estimate = quuux,
    std.err = quuuux)
```

Ready to make table.

```
foot <- as.data.table(foo) |>
    as_grouped_data(groups = c("Subset"),
        columns = c("change", "estimate", "std.err")) |>
    flextable() |>
    set_header_labels(change = "Change", estimate = "Estimate",
        std.err = "Std. Error") |>
    colformat_double(digits = 4)
foot
```

Table 8: Estimates of Change in Mean Fitness

Subset	Change	Estimate	Std. Error
CERA 2015-2016			
	total	0.3912	0.4744
	environmental	2.8458	0.5991

Table 8: Estimates of Change in Mean Fitness

Subset	Change	Estimate	Std. Error
	selection	0.1527	0.1044
	fftns	2.6975	1.2098
	residual	-2.6074	0.4694
CERA 2016-2017			
	total	-3.2343	0.4191
	environmental	-2.5649	0.5575
	selection	0.4823	0.0977
	fftns	0.9855	0.7613
	residual	-1.1517	0.3932
Grey Cloud Dunes 2015-2016			
	total	0.0559	0.1699
	environmental	-1.0985	0.1734
	selection	0.1683	0.0148
	fftns	0.8827	0.3254
	residual	0.9861	0.1334
Grey Cloud Dunes 2016-2017			
	total	1.7586	0.3778
	environmental	3.4862	0.4422
	selection	0.1247	0.0238
	fftns	3.8995	4.5955
	residual	-1.8523	0.5657
McCarthy Lake 2015-2016			
	total	-0.6403	0.4167
	environmental	-1.3173	0.3885
	selection	0.4043	0.0582
	fftns	3.3474	4.7430
	residual	0.2728	0.3775

Table 8: Estimates of Change in Mean Fitness

Subset	Change	Estimate	Std. Error
McCarthy Lake 2016-2017			
	total	-0.4672	0.2946
	environmental	-0.5562	0.2746
	selection	0.3486	0.0671
	fftns	2.0604	2.0346
	residual	-0.2597	0.1982

OK. Ship out to Microsoft Word.

```
foot <- set_caption(foot, "Estimates of Change in Mean Fitness")
save_as_docx(foot, path = "table3.docx")
```

13 Plotting the Decomposition, Try Two

We make more plots and output them as separate PDF files for the paper. This time no arrows.

13.1 Six Different Plots

```
crit <- qnorm(0.975)
crit
```

```
## [1] 1.959964
```

```
pdf("plots-six.pdf")
par(mar = c(3, 4, 4, 0) + 0.1)
par(mfrow = c(3, 2))
for (site in names(fout)) {
    for (year in names(fout[[1]])) {
    prev.year <- as.character(as.numeric(year) - 1)
        lab <- paste0(site.translate[site], " ", prev.year, "-", year)
        foo <- fout[[site]][[year]]
        errbar(1:4,
            foo$estimate,
            foo$estimate + crit * foo$std.err.,
            foo$estimate - crit * foo$std.err.,
        main = lab, axes = FALSE,
        ylab = "change in mean fitness", xlab = "",
        xlim = c(0.5, 4.5))
    box()
    axis(side = 2)
    axis(side = 1, at = 1:4, tick = FALSE, labels = names(foo$estimate))
    abline(h = 0)
    }
}
```

References

Geyer, C. J. and Shaw, R. G. (2008) *Commentary on Lande-Arnold analysis*. 670. School of Statistics, University of Minnesota. Available at: https://hdl.handle.net/11299/56218.

Geyer, C. J. and Shaw, R. G. (2013) *Aster models with random effects and additive genetic variance for fitness*. 696. School of Statistics, University of Minnesota. Available at: https://hdl.handle.net/11299/152355.

Geyer, C. J., Ridley, C. E., Latta, R. G., et al. (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., Kulbaba, M. W., Sheth, S. N., et al. (2022) Correction for Kulbaba et al. (2019). *Evolution*, **76**, 3074. DOI: 10.1111/evo.14607.

Kulbaba, M. W., Sheth, S. N., Pain, R. E., et al. (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.

Lande, R. and Arnold, S. J. (1983) The measurement of selection on correlated characters. *Evolution*, **37**, 1210–1226. DOI: 10.2307/2408842.

Shaw, R. G., Geyer, C. J., Wagenius, S., et al. (2008) Unifying life history analysis for inference of fitness and population growth. *American Naturalist*, **172**, E35–E47. DOI: 10.1086/588063.