The Aster Package Tutorial

Charles J. Geyer

December 7, 2025

1 Preliminaries

1.1 Library and Data

```
> library(aster)
> packageDescription("aster")$Version
```

```
[1] "1.3-7"
```

```
> data(echinacea)
```

That's our package and the dataset for our examples.

1.2 Digression on Data Entry

The simplest way to enter your own data into R is to construct a plain text file that is a white-space-separated matrix of numbers with column headings. (Note this means not touched by Microsoft Word or any so-called word processors that are incapable of not garbaging up plain text with a lot of unwanted formatting junk: if on Microsoft Windows, use Microsoft Notepad.)

An example is the file `"echinacea.txt"` found in the `data` directory of the package installation.

This file could read into R as follows

```
echinacea <- read.table("echinacea.txt", header = TRUE)
```

In fact that's what R did when loading the library.

If you have been so foolish as to give your data to Microsoft Excel, you can ask it to give it back by writing out in CSV (Microsoft comma separated values format), and it may give you back something usable (or it may not: dates are expecially problematic, for some reason). If the output is called `echinacea.csv` then

```
echinacea <- read.csv("echinacea.csv")
```

will read it into R.

1.3 Digression on Data Frames

In either case the R object `echinacea` created by the `data` statement or a `read.table` or `read.csv` command is what R calls a *data frame*. It prints like a matrix but is really a list of vectors of the same length. The command

```
> names(echinacea)
```

```
 [1] "hdct02" "hdct03" "hdct04" "pop"    "ewloc"  "nsloc"  "ld02"
 [8] "fl02"   "ld03"   "fl03"   "ld04"   "fl04"
```

shows the names of these (vector) variables. The names are the column headings from the input file. Any variables having character values have been coerced to what R calls *factors*. To see what kind of variable each is we can do

```
> sapply(echinacea, class)
```

```
   hdct02    hdct03    hdct04       pop     ewloc     nsloc      ld02
"integer" "integer" "integer"  "factor" "integer" "integer" "integer"
     fl02      ld03      fl03      ld04      fl04
"integer" "integer" "integer" "integer" "integer"
```

In the `echinacea` data set, the variables with numbers in the names are the columns of the response matrix of the aster model.

- The variables `ld0`x (where x is a digit) are the survival indicator variables for year 200x (one for alive, zero for dead).
- The variables `fl0`x are the flowering indicator variables (one for any flowers, zero for none).
- The variables `hdct0`x are the inflorescence (flower head) count variables (number of flower heads).

The variables without numbers are other predictors.

- The variables `ewloc` and `nsloc` are spatial, east-west and north-south location, respectively.

- The variable `pop` is the remnant population of origin of the plant, so plants with different values of `pop` may be more genetically diverse than those with the same values of `pop`.

The point of recording the data about flowers in two variables `fl0`x and `hdct0`x for any given x when the single variable `hdct0`x contains the same information (because `fl0`$x = 0$ if and only if `hdct0`$x = 0$) is because there is no good, simple statistical model for `hdct0`x by itself.

As we shall see, it makes sense to model the conditional distribution of `fl0`x given `ld0`$x = 1$ as Bernoulli and it makes sense to model the conditional distribution of `hdct0`x given `fl0`$x = 1$ as Poisson conditioned on being nonzero. But it would not make sense to model `hdct0`x given `ld0`$x = 1$ as Poisson. The case of zero flowers is special and must be modeled separately.

1.4 Digression on Reshape

Standard regression-like modeling requires that the "response" be a vector. The "response" ("data" would be a better name since it plays two roles: $X_{ip(j)}$ is the sample size for X_{ij}) is a matrix. If we are going to use any of the R apparatus for doing regression-like modeling, in particular if we are going to use the R formula mini-language that allows model specifications like `y ~ x1 + x2`, then we need to make the response (the `y` in the formula) a vector.

There is an R function `reshape` that does this

```
> vars <- c("ld02", "ld03", "ld04", "fl02", "fl03", "fl04",
+     "hdct02", "hdct03", "hdct04")
> redata <- reshape(echinacea, varying = list(vars),
+      direction = "long", timevar = "varb", times = as.factor(vars),
+      v.names = "resp")
> names(redata)

[1] "pop"   "ewloc" "nsloc" "varb"  "resp"  "id"
```

making a new data frame (in this case `redata`) that has new variables (all of the same length because they are in the same data frame)

`resp` which contains all of the data in the variables indicated by the string `vars` packed into a single vector, the name `resp` being given by the `v.names` argument of the `reshape` function (you could make it anything you like),

`varb` which indicates which original variable the corresponding element of `resp` came from, the name being given by the `timevar` argument of the `reshape` function, and

`id` which indicates which original individual the corresponding element of `resp` came from, the name being given by the `idvar` argument of the `reshape` function (which we didn't specify, accepting the default value `"id"`).

Warning The variable `vars` must have the variables listed with parent nodes before child nodes. This doesn't make sense now, but is nessary to satisfy condition (∗) on page 6.

Warning The variable `varb` in the data frame produced by the `reshape` function (here called `redata`) must be a factor.

```
> class(redata$varb)

[1] "factor"

> levels(redata$varb)

[1] "fl02"   "fl03"   "fl04"   "hdct02" "hdct03" "hdct04" "ld02"
[8] "ld03"   "ld04"
```

The `redata` command used above accomplishes this. If the argument

```
times = as.factor(vars)
```

of the `redata` function is omitted in the call, then this will not happen and all of the following analysis will be wrong.

Let's look at an example

```
> redata[42, ]

        pop ewloc nsloc varb resp id
42.ld02 SPP    -3     7 ld02    1 42
```

This says that the 42nd row of the data frame `redata` has response value 1. This is the response for the `ld02` original variable (node of the graphical model) and for individual 42 found in row 42 of the original data frame `echinacea`. The other variables `pop`, `ewloc`, and `nsloc` are what they were in this row of `echinacea` (for this original individual) and are duplicated as necessary in other rows of `redata`.

2 Unconditional Aster Models

Now we are ready to begin discussion of Aster models.

2.1 More Preliminaries

Then more details. We must define a few more variables that determine the structure of the aster model we intend to work with. The variable `resp` contains only the random data. There is also the non-random data on the root nodes of the dependence graph of the model. For reasons having to do with the way the R formula mini-language works, we usually want this in the same data frame as `resp`

```
> redata <- data.frame(redata, root = 1)
> names(redata)

[1] "pop"   "ewloc" "nsloc" "varb"  "resp"  "id"    "root"
```

This adds a variable `root` to the data frame and makes all its values one (including for non-root nodes, but those values are ignored by all `aster` package functions).

We also need two vectors of the same length as the number of nodes in the graph (so they cannot go in the data frame `redata`), call that length `nnode`

```
> pred <- c(0, 1, 2, 1, 2, 3, 4, 5, 6)
> fam <- c(1, 1, 1, 1, 1, 1, 3, 3, 3)
> sapply(fam.default(), as.character)[fam]

[1] "bernoulli"
[2] "bernoulli"
[3] "bernoulli"
[4] "bernoulli"
[5] "bernoulli"
[6] "bernoulli"
[7] "truncated.poisson(truncation = 0)"
[8] "truncated.poisson(truncation = 0)"
[9] "truncated.poisson(truncation = 0)"
```

`pred` specifies the predecessor structure of the graph. With the original variables indexed from 1 to `nnode`, in the order that they appear in the variable `vars` defined above and supplied as the `varying` argument

to the `reshape` function, `pred[j]` gives the index of the predecessor of node `j`. In order to make it obvious that `pred` specifies an acyclic graph, we require

$$\texttt{all(pred < seq(along = pred))} \qquad (*)$$

hold. If `pred[j] == 0` this indicates `j` indexes a root node (no predecessor).

`fam` specifies the one-parameter exponential families for the nodes: `fam[j]` gives the index of the family for node `j`, the index being for the list of family specifications that is supplied to model fitting functions, the default being the list returned by `fam.default()`.

Comment Using an index vector rather than a character vector for `fam` is definitely not the R way. But having decided to make `pred` an index vector, the same seemed natural for `fam`.

The reason for using an index vector for `pred` is so the condition $(*)$ makes sense. The original variables must be related to their order in the data frame `redata` in any case.

This may change in the future, but that's the way it is for now.

2.2 Fitting a Model

We are finally ready to fit a model.

```
> aout1 <- aster(resp ~ varb + nsloc + ewloc + pop,
+     pred, fam, varb, id, root, data = redata)
> summary(aout1, show.graph = TRUE)

Call:
aster.formula(formula = resp ~ varb + nsloc + ewloc + pop, pred = pred,
    fam = fam, varvar = varb, idvar = id, root = root, data = redata)


Graphical Model:
 variable predecessor family
 ld02     root        bernoulli
 ld03     ld02        bernoulli
 ld04     ld03        bernoulli
 fl02     ld02        bernoulli
```

```
 fl03     ld03        bernoulli
 fl04     ld04        bernoulli
 hdct02   fl02        truncated.poisson(truncation = 0)
 hdct03   fl03        truncated.poisson(truncation = 0)
 hdct04   fl04        truncated.poisson(truncation = 0)

             Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.888113   0.184006  -4.827 1.39e-06 ***
varbfl03    -0.329497   0.265611  -1.241  0.21478
varbfl04    -0.355326   0.241242  -1.473  0.14078
varbhdct02   1.174051   0.249520   4.705 2.54e-06 ***
varbhdct03   1.191939   0.200166   5.955 2.60e-09 ***
varbhdct04   1.704852   0.185538   9.189  < 2e-16 ***
varbld02    -0.122959   0.311876  -0.394  0.69339
varbld03     1.638747   0.392935   4.171 3.04e-05 ***
varbld04     4.084487   0.330810  12.347  < 2e-16 ***
nsloc        0.013684   0.001733   7.895 2.91e-15 ***
ewloc        0.006003   0.001727   3.475  0.00051 ***
popEriley   -0.052273   0.053504  -0.977  0.32857
popLf       -0.046125   0.056892  -0.811  0.41751
popNWLF     -0.061712   0.051028  -1.209  0.22652
popNessman  -0.124857   0.070894  -1.761  0.07821 .
popSPP       0.036007   0.054506   0.661  0.50887
popStevens  -0.084581   0.054256  -1.559  0.11901
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The `show.graph = TRUE` makes the table about the graph structure in the printout. You always want to look at that carefully once, since it tells you whether you have gotten it right. But (unless you change `redata`, `pred`, or `fam`) it won't change thereafter. Hence the default is to not print it.

Interpretation. From looking at the "signif. stars" it looks like `pop` is not significant. But you're not supposed to use "signif. stars" that way. Any use of more than one "signif. star" per model fit is *multiple testing without correction* and may be *highly misleading*. Wise users put

```
> options(show.signif.stars = FALSE)
```

in their `.Rprofile` file so they aren't even tempted to foolishness by them. The R default is to print them, so that's what `summary.aster` does too.

2.3 Model Comparison

The right way to compare models is with a likelihood ratio test. To do that, you must fit two models. So here's another.

```
> aout2 <- aster(resp ~ varb + nsloc + ewloc,
+     pred, fam, varb, id, root, data = redata)
```

Now we can compare them with

```
> anova(aout2, aout1)

Analysis of Deviance Table

Model 1: resp ~ varb + nsloc + ewloc
Model 2: resp ~ varb + nsloc + ewloc + pop
  Model Df Model Dev Df Deviance P(>|Chi|)
1       11   -2747.9
2       17   -2734.4  6   13.515   0.03554
```

So much for making inferences from a bunch of signif. stars! Not one of the signif. stars for the `pop` dummy variables in the printout for `aout1` was anywhere near as significant as the likelihood ratio test p-value here. See what we mean?

Warning (copied from the `anova.aster` help page)

> The comparison between two or more models . . . will only be valid if they are (1) fitted to the same dataset, (2) models are nested, (3) models are of the same type (all conditional or all unconditional), (4) have the same dependence graph and exponential families. None of this is currently checked.

You're not likely to botch (1) or (4). Just make sure you use the same data frame and the same `pred` and `fam`. If you're a little careful, you won't botch (3). We haven't done conditional models yet (see Section 4), but when we do, those models shouldn't be mixed with unconditional models. It's mixing apples and oranges. The hard condition to always obey is (2). One must be sure that the column space of the model matrix of the big model (`aout1$modmat`) contains the column space of the model matrix of the small model (`aout2$modmat`), but that is much easier checked by the computer than by you.

Comment So perhaps the computer should check all this, despite none of the R `anova` functions doing so. It would take time, enough for one linear regression per column of the model matrix of the small model, but it would add safety. The R/C/Unix way (a. k. a., worse is better) is to not check and let the user worry about it.

Anyway (as the warning says) `anova.aster` and `anova.asterlist` currently do not check. The user must make sure the models are nested (otherwise the comparison is theoretically rubbish).

2.4 Models Based on Pseudo-Covariates

In a sense the "covariate" variable `varb` is already a pseudo-covariate. It's not a measured variable. It's an artifice we use to overcome the limitations of the R formula mini-language insisting that the "response" be a vector, rather than a matrix with heterogenous columns (like it really is).

In this section we use more artifice. The variables of interest, the best surrogates of Darwinian fitness, are the `hdct` variables. We want to look that the interaction of `pop` with those variables. So first we have to create the relevant dummy variable.

```
> hdct <- grep("hdct", as.character(redata$varb))
> hdct <- is.element(seq(along = redata$varb), hdct)
> redata <- data.frame(redata, hdct = as.integer(hdct))
> names(redata)
```

```
[1] "pop"   "ewloc" "nsloc" "varb"  "resp"  "id"    "root"  "hdct"
```

This is tricky. The variable `redata$varb` is a factor, which means it is really numeric (an index vector) under the hood, even though it prints as a character string. The expression `as.character(redata$varb)` turns these numeric values into strings. Then the `grep` function returns the *indices* of the elements of `varb` whose string forms contain `"hdct"`. The next statement converts these to `TRUE` or `FALSE`. The third statement converts these to one and zero, and makes them the variable `hdct` in the data frame `redata`. You might not think of such an R-ish way to do this, but if you want to use such a variable in modeling, you must create it somehow.

Now we can fit an interaction term.

```
> aout3 <- aster(resp ~ varb + nsloc + ewloc + pop * hdct,
+     pred, fam, varb, id, root, data = redata)
> summary(aout3)
```

```
Call:
aster.formula(formula = resp ~ varb + nsloc + ewloc + pop * hdct,
    pred = pred, fam = fam, varvar = varb, idvar = id, root = root,
    data = redata)

                Estimate Std. Error z value Pr(>|z|)
(Intercept)     -1.042324   0.216993  -4.804 1.56e-06
varbfl03        -0.328716   0.264185  -1.244 0.213403
varbfl04        -0.339642   0.239818  -1.416 0.156702
varbhdct02       1.455401   0.323007   4.506 6.61e-06
varbhdct03       1.473010   0.287024   5.132 2.87e-07
varbhdct04       1.984495   0.277131   7.161 8.02e-13
varbld02        -0.126333   0.312215  -0.405 0.685747
varbld03         1.620832   0.393076   4.123 3.73e-05
varbld04         4.049220   0.330949  12.235  < 2e-16
nsloc            0.013538   0.001727   7.838 4.59e-15
ewloc            0.005990   0.001724   3.474 0.000513
popEriley        0.332627   0.152129   2.186 0.028780
popLf            0.318859   0.162391   1.964 0.049584
popNWLF          0.017505   0.142818   0.123 0.902449
popNessman       0.211191   0.187665   1.125 0.260435
popSPP           0.168390   0.156462   1.076 0.281822
popStevens      -0.083259   0.150749  -0.552 0.580741
popEriley:hdct  -0.717275   0.259529  -2.764 0.005714
popLf:hdct      -0.678378   0.278899  -2.432 0.015001
popNWLF:hdct    -0.127151   0.241487  -0.527 0.598518
popNessman:hdct -0.673876   0.354761  -1.900 0.057496
popSPP:hdct     -0.238292   0.258664  -0.921 0.356924
popStevens:hdct  0.027072   0.256538   0.106 0.915956

Original predictor variables dropped (aliased)
     hdct
```

Comment The `dropped (aliased)` means just what it says. The R formula mini-language does not deal correctly with pseudo-covariates like this and thinks it should put (a dummy variable for) `hdct` in the model, but that is aliased with the sum of (the dummy variables for) `varbhdct02`, `varbhdct03`, and `varbhdct04`, so the `aster` function drops it, but not silently (so the user is not surprised).

It turns out this is not the model we wanted to fit. We didn't want population main effects, `popEriley` and so forth, *in addition to* the population interaction effects `popEriley:hdct` and so forth. We only wanted `pop` to have effects at the `hdct` level. Here's how we do that

```
> aout4 <- aster(resp ~ varb + nsloc + ewloc + pop * hdct - pop,
+     pred, fam, varb, id, root, data = redata)
> summary(aout4)

Call:
aster.formula(formula = resp ~ varb + nsloc + ewloc + pop * hdct -
    pop, pred = pred, fam = fam, varvar = varb, idvar = id, root = root,
    data = redata)

                 Estimate Std. Error z value Pr(>|z|)
(Intercept)     -0.922313   0.178183  -5.176 2.26e-07
varbfl03        -0.328488   0.265142  -1.239 0.215377
varbfl04        -0.349519   0.240775  -1.452 0.146601
varbhdct02       1.267123   0.258014   4.911 9.06e-07
varbhdct03       1.284265   0.210824   6.092 1.12e-09
varbhdct04       1.797630   0.197099   9.120  < 2e-16
varbld02        -0.144176   0.311742  -0.462 0.643733
varbld03         1.621541   0.392815   4.128 3.66e-05
varbld04         4.070145   0.330729  12.307  < 2e-16
nsloc            0.013600   0.001729   7.867 3.64e-15
ewloc            0.006473   0.001725   3.753 0.000174
popEriley:hdct  -0.173902   0.089753  -1.938 0.052676
popLf:hdct      -0.157706   0.096580  -1.633 0.102490
popNWLF:hdct    -0.109268   0.083325  -1.311 0.189740
popNessman:hdct -0.315681   0.139822  -2.258 0.023962
popSPP:hdct      0.023541   0.086552   0.272 0.785630
popStevens:hdct -0.127403   0.089518  -1.423 0.154677

Original predictor variables dropped (aliased)
     hdct
```

and here's the ANOVA (analysis of deviance, log likelihood ratio test) table for these models

```
> anova(aout2, aout4, aout3)
```

```
Analysis of Deviance Table

Model 1: resp ~ varb + nsloc + ewloc
Model 2: resp ~ varb + nsloc + ewloc + pop * hdct - pop
Model 3: resp ~ varb + nsloc + ewloc + pop * hdct
  Model Df Model Dev Df Deviance P(>|Chi|)
1       11   -2747.9
2       17   -2731.9  6   15.947 0.0140395
3       23   -2707.9  6   24.043 0.0005129
```

Comment When I did this the first time, I made the mistake of trying to put model `aout1` between `aout2` and `aout4`, but that's the mistake we were warned about above! The models `aout2` and `aout4` are not nested! (This is obvious when you see they have the same degrees of freedom, and even more so when you think about the dummy variables they contain.) The only warning I got was about using zero degrees of freedom in a chi-square, not a warning about non-nested models. Be careful! (This warning goes for all uses of `anova` in R. It's not only a problem with `anova.aster`.)

Comment That's all for modeling of unconditional aster models. Note that these are not the spatial effects that were judged the best fitting in the paper. See the tech report (`http://www.stat.umn.edu/geyer/aster/`) for details.

3 Prediction

Once you have an aster model, it is a natural question to ask what it means. The `anova` tests give some information about this (which models fit better and which fit worse), but there is a lot more to statistics than hypothesis tests.

The other standard thing to do with regression-like models is to make confidence intervals for parameters. For reasons that don't make any sense in the context of aster models, the function that does this is a method of the `predict` generic function (the reason is that for linear models only, this function does both confidence intervals and so-called prediction intervals).

With aster models we have a larve variety of parameters to "predict". Every aster model comes with five different parameterizations

$\boldsymbol{\beta}$ the regression coefficient vector,

$\boldsymbol{\theta}$ the conditional canonical parameter vector,

$\boldsymbol{\varphi}$ the unconditional canonical parameter vector,

$\boldsymbol{\xi}$ the conditional mean value parameter vector, and

$\boldsymbol{\tau}$ the unconditional mean value parameter vector.

3.1 Regression Coefficients

The package provides no special support for confidence intervals about regression coefficients because of the dictum

> Regression coefficients are meaningless. Only probabilities and expectations are meaningful.

In further support of the meaninglessness of regression coefficients note that two models specified by different model matrices that have the same *column spaces* are essentially the same model. The MLE regression coefficients will be different, but the MLE of $\boldsymbol{\theta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\xi}$, and $\boldsymbol{\tau}$ are identical for both models, as are predicted probabilities of all events and predicted expectations of all random variables. Once you understand that, how can you *reify* regression coefficients?

3.2 Using the Summary

Nevertheless, now that we have had our tendentious rant, we do tell you how to make confidence intervals for betas. The summary function gives standard errors, so one way is

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

> fred <- summary(aout4)
> dimnames(fred$coef)

[[1]]
 [1] "(Intercept)"     "varbfl03"         "varbfl04"
 [4] "varbhdct02"      "varbhdct03"       "varbhdct04"
 [7] "varbld02"        "varbld03"         "varbld04"
[10] "nsloc"           "ewloc"            "popEriley:hdct"
[13] "popLf:hdct"      "popNWLF:hdct"     "popNessman:hdct"
[16] "popSPP:hdct"     "popStevens:hdct"
```

```
[[2]]
[1] "Estimate"   "Std. Error" "z value"    "Pr(>|z|)"
```

```
> fred$coef["popEriley:hdct", "Estimate"] +
+ c(-1, 1) * crit * fred$coef["popEriley:hdct", "Std. Error"]
```

```
[1] -0.349813543  0.002010091
```

This gives an asymptotic (large sample) 95% confidence interval for the unknown true regression coefficient for the dummy variable `popEriley:hdct` (assuming this model is correct!)

3.3 Using Fisher Information

Those who know master's level theoretical statistics, know the standard error here is based on inverse Fisher information, which is found in the output of `aster`. So another way to do the same interval is

```
> aout4$coef[12] + c(-1, 1) * crit * sqrt(solve(aout4$fish)[12, 12])
```

```
[1] -0.349813543  0.002010091
```

3.4 Using Fisher Information (Continued)

The reason for even mentioning Fisher information is that it is essential if you want to make any other intervals for betas other than the ones `summary.aster` helps with. For example, suppose you want an interval for $\beta_{12} - \beta_{13}$. How do you do that?

```
> inv.fish.info <- solve(aout4$fish)
> (aout4$coef[12] - aout4$coef[13]) + c(-1, 1) * crit *
+ sqrt(inv.fish.info[12, 12] + inv.fish.info[13, 13] - 2 * inv.fish.info[12, 13])
```

```
[1] -0.1726329  0.1402422
```

If that doesn't make sense, you'll have to review your master's level theory notes. We can't explain everything. But fortunately, if you understand the meaninglessness of regression coefficients, you'll never want to do that anyway.

3.5 Canonical Parameters

Canonical parameters are only slightly less meaningless than regression coefficients. They still aren't probabilities and expectations that have real world meaning. However, `predict.aster` does give support for "predicting" canonical parameters.

There are, however a huge number of canonical parameters, one for each individual and node of the graphical model (same goes for mean value parameters). There are two ways we simplify the situation (both are standard in doing regression "predictions")

- predict only for one (or a few) hypothetical or real "new" individuals and
- predict a linear functional of the parameter.

For the former we "predict" using a new model matrix whose entries refer to the "new" individuals rather than the "old" ones we used to fit the data. For the latter we predict $\mathbf{A}'\boldsymbol{\zeta}$ where $\mathbf{A}$ is a fixed, known matrix and $\boldsymbol{\zeta}$ is the parameter we want to estimate (any of $\boldsymbol{\theta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\xi}$, or $\boldsymbol{\tau}$).

3.5.1 New Data

We construct new data for "typical" individuals (having zero-zero geometry) in each population.

```
> newdata <- data.frame(pop = levels(echinacea$pop))
> for (v in vars)
+     newdata[[v]] <- 1
> newdata$root <- 1
> newdata$ewloc <- 0
> newdata$nsloc <- 0
```

We are using bogus data $x_{ij} = 1$ for all i and j. The predictions for $\boldsymbol{\theta}$ and $\boldsymbol{\varphi}$ do not depend on new data anyway. Predictions for $\boldsymbol{\tau}$ depend only on the new `root` data but not on the other data variables.

Predictions for $\boldsymbol{\xi}$ do depend only on all the new data, but for hypothetical individuals we have no data to give them! This is an odd aspect of aster models, that data x_{ij} plays the role of the response and also of a predictor when it appears as $x_{ip(j)}$. The prediction section of the paper explains why we usually want this hypothetical data to be all ones.

```
> renewdata <- reshape(newdata, varying = list(vars),
+      direction = "long", timevar = "varb", times = as.factor(vars),
+      v.names = "resp")
> hdct <- grep("hdct", as.character(renewdata$varb))
> hdct <- is.element(seq(along = renewdata$varb), hdct)
> renewdata <- data.frame(renewdata, hdct = as.integer(hdct))
> names(redata)

[1] "pop"   "ewloc" "nsloc" "varb"  "resp"  "id"    "root"  "hdct"

> names(renewdata)

[1] "pop"   "root"  "ewloc" "nsloc" "varb"  "resp"  "id"    "hdct"
```

3.5.2 Linear Functional Matrix

The functional of mean value parameters we want is *total head count*, since this has the biological interpretation of the best surrogate of fitness measured in this data set. A biologist (at least an evolutionary biologist) is interested in the "predecessor variables" of head count only insofar as they contribute to head count. Two sets of parameter values that "predict" the same expected total head count (over the three years the data were collected) have the same contribution to fitness. So that is the "prediction" (really functional of mean value parameters) we "predict."

```
> nind <- nrow(newdata)
> nnode <- length(vars)
> amat <- array(0, c(nind, nnode, nind))
> for (i in 1:nind)
+     amat[i , grep("hdct", vars), i] <- 1
```

This says that the i-th component of the predicted linear functional is the sum of the variables having `"hdct"` in their names for the i-th individual (in the new hypothetical data `newdata`). The linear functional is a simple sum because the elements of `amat` are all zero or one.

This is a little tricky, so let's take it one step at a time. We are trying to form the linear functional $\mathbf{A}'\boldsymbol{\tau}$, which written out in full is the vector with k-th component

$$\sum_{i \in I_{\text{new}}} \sum_{j \in J} a_{ijk} \tau_{ij}$$

where a_{ijk} are the elements of the array $\mathbf{A}$ and τ_{ij} are the elements of the matrix of unconditional mean value parameters $\boldsymbol{\tau}$ and where I_{new} is the index

set for the "new hypothetical" individuals in `newdata`. Since there is one such "new hypothetical" individual for each population, I_{new} indexes populations *in this particular example*.

For such an individual, for an $i \in I_{\text{new}}$, we want the corresponding component of $\mathbf{A}'\boldsymbol{\tau}$ to be that individual's total flower head count, the sum of the τ_{ij} such that j is a head count variable. Hence *in this particular example* we want the dimension of $\mathbf{A}'\boldsymbol{\tau}$ to be the same as the number of individuals (meaning i and k run over the same index set I_{new}). The way we make $\mathbf{A}'\boldsymbol{\tau}$ be this sum is to set the components of $a_{i,\cdot,i}$ be one for the terms we want in the sum and zero for the terms we don't want in the sum, and that's exactly what the R code above does.

Note, that in general i and k will run over different index sets and that in general there is no reason why the components a_{ijk} must be zero or one. How one constructs a_{ijk} varies a great deal from application to application. This example only illustrates one very particular special case.

3.5.3 Prediction

So we are finally ready to make a "prediction"

```
> foo <- predict(aout4, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = amat,
+     parm.type = "canon")
> bar <- cbind(foo$fit, foo$se.fit)
> dimnames(bar) <- list(as.character(newdata$pop), c("Estimate", "Std. Error"))
> print(bar)

         Estimate Std. Error
AA      1.5820795  0.2478499
Eriley  1.0603744  0.1972253
Lf      1.1089604  0.2239097
NWLF    1.2542756  0.1679685
Nessman 0.6350351  0.3800243
SPP     1.6527033  0.1746258
Stevens 1.1998709  0.1945909
```

Since the default `model.type` is `"unconditional"`, these are (linear functionals of) the unconditional canonical parameters $\boldsymbol{\varphi}$.

And for our next trick (neither of these are motivated by a particular scientific question—we're just illustrating the use of `predict.aster`).

```
> foo <- predict(aout4, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = amat,
+     parm.type = "canon", model.type = "cond")
> bar <- cbind(foo$fit, foo$se.fit)
> dimnames(bar) <- list(as.character(newdata$pop), c("Estimate", "Std. Error"))
> print(bar)

         Estimate Std. Error
AA      1.5820795  0.2478499
Eriley  1.0603744  0.1972253
Lf      1.1089604  0.2239097
NWLF    1.2542756  0.1679685
Nessman 0.6350351  0.3800243
SPP     1.6527033  0.1746258
Stevens 1.1998709  0.1945909
```

These are (linear functionals of) the conditional canonical parameters $\boldsymbol{\theta}$.

Woof! When I saw this the first time, I spent over an hour trying to track down the bug in `predict.aster` that gives the same predictions for two different types of parameters. But it's not a bug. The canonical parameters for "leaf" nodes (no successors) *are* the same!

Let's try the `"ld"` level (mortality)

```
> bmat <- array(0, c(nind, nnode, nind))
> for (i in 1:nind)
+     bmat[i , grep("ld", vars), i] <- 1

> foo <- predict(aout4, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = bmat,
+     parm.type = "canon")
> bar <- cbind(foo$fit, foo$se.fit)
> dimnames(bar) <- list(as.character(newdata$pop), c("Estimate", "Std. Error"))
> print(bar)

          Estimate Std. Error
AA      -0.6851641  0.1271524
Eriley  -0.6851641  0.1271524
Lf      -0.6851641  0.1271524
NWLF    -0.6851641  0.1271524
Nessman -0.6851641  0.1271524
SPP     -0.6851641  0.1271524
Stevens -0.6851641  0.1271524
```

```
> foo <- predict(aout4, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = bmat,
+     parm.type = "canon", model.type = "cond")
> bar <- cbind(foo$fit, foo$se.fit)
> dimnames(bar) <- list(as.character(newdata$pop), c("Estimate", "Std. Error"))
> print(bar)

        Estimate Std. Error
AA      7.944257  0.6177680
Eriley  6.928158  0.4348137
Lf      7.009798  0.4758742
NWLF    7.268865  0.4173700
Nessman 6.307471  0.5759345
SPP     8.108078  0.4831169
Stevens 7.169185  0.4481993
```

Now quite different! And we have another puzzle. Why, according to aster model theory is it the right thing that all of the ξ_{ij} are the same where j is for an `"ld"` node? Just look at the model: we don't have an interaction of `ld` and `pop`.

```
> foo <- predict(aout3, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = bmat,
+     parm.type = "canon")
> bar <- cbind(foo$fit, foo$se.fit)
> dimnames(bar) <- list(as.character(newdata$pop), c("Estimate", "Std. Error"))
> print(bar)

           Estimate Std. Error
AA      -1.04898992  0.4036746
Eriley  -0.05110772  0.2496746
Lf      -0.09241159  0.3025186
NWLF    -0.99647496  0.1949520
Nessman -0.41541594  0.4080958
SPP     -0.54382117  0.2825132
Stevens -1.29876764  0.2422731
```

If instead we use the model `aout3`, which does have such an interaction (the interaction is with non-`hdct`, but that includes `ld`), then the ξ_{ij} are different.

What this all proves other than that aster models are complicated, I don't know. The point is just to give some examples. We don't think scientists

should want to "predict" canonical parameters much (except perhaps just to see what's going on under the hood of some model).

3.6 Mean Value Parameters

To repeat our mantra

> Only probabilities and expectations are meaningful.

The parameters that `predict.aster` can predict and that are meaningful (according to our mantra) are *mean value parameters*. As with everything else, there are two kinds, conditional and unconditional ($\boldsymbol{\xi}$ and $\boldsymbol{\tau}$, respectively).

Let's try that.

```
> pout3 <- predict(aout3, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = amat)
> pout4 <- predict(aout4, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = amat)
```

Figure 1 is produced by the following code

```
> popnames <- as.character(newdata$pop)
> fit3 <- pout3$fit
> fit4 <- pout4$fit
> i <- seq(along = popnames)
> foo <- 0.1
> y4top <- fit4 + crit * pout4$se.fit
> y4bot <- fit4 - crit * pout4$se.fit
> y3top <- fit3 + crit * pout3$se.fit
> y3bot <- fit3 - crit * pout3$se.fit
> plot(c(i - 1.5 * foo, i - 1.5 * foo, i + 1.5 * foo, i + 1.5 * foo),
+     c(y4top, y4bot, y3top, y3bot), type = "n", axes = FALSE,
+     xlab = "", ylab = "")
> segments(i - 1.5 * foo, y4bot, i - 1.5 * foo, y4top)
> segments(i - 2.5 * foo, y4bot, i - 0.5 * foo, y4bot)
> segments(i - 2.5 * foo, y4top, i - 0.5 * foo, y4top)
> segments(i - 2.5 * foo, fit4, i - 0.5 * foo, fit4)
> segments(i + 1.5 * foo, y3bot, i + 1.5 * foo, y3top, lty = 2)
> segments(i + 2.5 * foo, y3bot, i + 0.5 * foo, y3bot)
> segments(i + 2.5 * foo, y3top, i + 0.5 * foo, y3top)
> segments(i + 2.5 * foo, fit3, i + 0.5 * foo, fit3)
```

```
> axis(side = 2)
> title(ylab = "unconditional mean value parameter")
> axis(side = 1, at = i, labels = popnames)
> title(xlab = "population")
```

and appears on p. 22.

Figure 1 looks pretty much like the plot in the paper. It's a little different because the spatial part of the model (involving `ewloc` and `nsloc` is simpler).

These confidence intervals have real-world meaning. The parameters predicted are expected flower head count over the course of the experiment for "typical" individuals ("typical" spatial location smack dab in the middle because we centered the spatial coordinates to have median zero) at each of the populations. (BTW I have no idea why `axis` leaves off one of the population labels when there's plenty of room. To get this plot right for the paper I just edited the PostScript file and put in the missing label by hand. If you ask me, it's a bug. But the `axis` maintainer would probably say it's a feature.)

As we have seen, you can "predict" any kind of parameter ($\boldsymbol{\beta}$, $\boldsymbol{\theta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\xi}$, or $\boldsymbol{\tau}$) for any model. All of the examples so far have been unconditional aster models (FEF). But we could have done everything above if we had originally fit conditional aster models (CEF).

4 Conditional Aster Models

We could now repeat everything above *mutatis mutandis*. The only thing that would change is an argument `type = "conditional"` to any call to the `aster` function. Then everything else works exactly the same (of course the actual numerical results are different, but the other function calls to `summary.aster`, `anova.aster`, or `predict.aster` are meaningful without change).

Let's look at one example.

```
> cout4 <- aster(resp ~ varb + nsloc + ewloc + pop * hdct - pop,
+     pred, fam, varb, id, root, data = redata, type = "cond")
> pcout4 <- predict(cout4, varvar = varb, idvar = id, root = root,
+     newdata = renewdata, se.fit = TRUE, amat = amat)
```

Note that these are exactly the same as the commands that made `pout4` except for the extra argument `type = "cond"` in the `aster` function call. Thus we have fit a conditional aster model (CEF) but predicted exactly the same $\mathbf{A}'\boldsymbol{\tau}$ as in `pout4`.

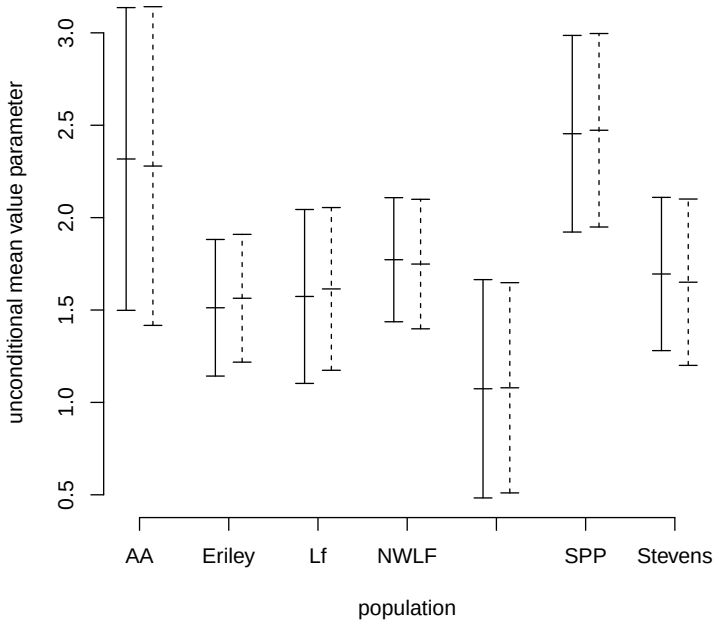

Figure 1: 95% confidence intervals for unconditional mean value parameter for fitness (sum of head count for all years) at each population for a "typical" individual having position zero-zero and having the parameterization of Model 4 (solid bar) or Model 3 (dashed bar). Tick marks in the middle of the bars are the center (the MLE).

Figure 2 compares these two "predictions" and is produced by the following code

```
> popnames <- as.character(newdata$pop)
> fit3 <- pcout4$fit
> fit4 <- pout4$fit
> i <- seq(along = popnames)
> foo <- 0.1
> y4top <- fit4 + crit * pout4$se.fit
> y4bot <- fit4 - crit * pout4$se.fit
> y3top <- fit3 + crit * pcout4$se.fit
> y3bot <- fit3 - crit * pcout4$se.fit
> plot(c(i - 1.5 * foo, i - 1.5 * foo, i + 1.5 * foo, i + 1.5 * foo),
+     c(y4top, y4bot, y3top, y3bot), type = "n", axes = FALSE,
+     xlab = "", ylab = "")
> segments(i - 1.5 * foo, y4bot, i - 1.5 * foo, y4top)
> segments(i - 2.5 * foo, y4bot, i - 0.5 * foo, y4bot)
> segments(i - 2.5 * foo, y4top, i - 0.5 * foo, y4top)
> segments(i - 2.5 * foo, fit4, i - 0.5 * foo, fit4)
> segments(i + 1.5 * foo, y3bot, i + 1.5 * foo, y3top, lty = 2)
> segments(i + 2.5 * foo, y3bot, i + 0.5 * foo, y3bot)
> segments(i + 2.5 * foo, y3top, i + 0.5 * foo, y3top)
> segments(i + 2.5 * foo, fit3, i + 0.5 * foo, fit3)
> axis(side = 2)
> title(ylab = "unconditional mean value parameter")
> axis(side = 1, at = i, labels = popnames)
> title(xlab = "population")
```

and appears on p. 24.

Now note that the two types of intervals (solid and dashed) are wildly different. Of course, they *should* be wildy different because the two types of models, one with $\boldsymbol{\varphi} = \mathbf{M}\boldsymbol{\beta}$ and the other with $\boldsymbol{\theta} = \mathbf{M}\boldsymbol{\beta}$, are radically different. There is no reason for the two types of predictions (of exactly the same thing) based on two radically different models should be similar. And they aren't. So we are only seeing what we should expect to see, and this doesn't prove anything about anything (except for the trivial fact that FEF and CEF aster models are different).

Comment We haven't really done justice to conditional models. Since this is a tutorial, we don't have to. As we said (twice), everything is the same except for `type = "cond"` in the appropriate place.

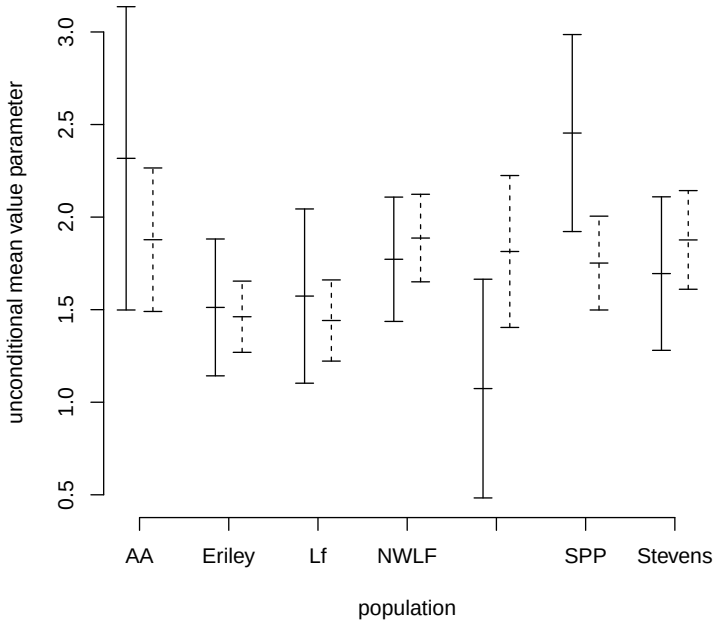

Figure 2: 95% confidence intervals for unconditional mean value parameter for fitness (sum of head count for all years) at each population for a "typical" individual having position zero-zero and having the parameterization of Model 4, either an unconditional model (solid bar) or conditional model (dashed bar). Tick marks in the middle of the bars are the center (the MLE).

And this is not the place to explain why one would want to use a conditional or unconditional model. And, to be honest, we don't know. We know, and explain in the paper, what each one does and why

> Conditional aster models are simple algebraically, but complicated statistically.
>
> Unconditional aster models are complicated algebraically, but simple statistically.

But we have to admit that these arguments from theoretical statistics may cut no ice with scientists, even if understood. It all depends on what particular scientific questions one is trying to answer. Thus we have written the `aster` package to be completely even-handed (except for defaults) with respect to conditional and unconditional (models or parameters). Whatever can be done with one kind can be done with the other. So use whatever you think right.

5 Simulation

One function we haven't covered is `raster` which simulates aster models. Let's check the coverage of some of our so-called 95% confidence intervals.

We start with another use of `predict`. The `raster` function wants $\boldsymbol{\theta}$ or, to be more precise, here we use $\hat{\boldsymbol{\theta}}$.

```
> theta.hat <- predict(aout4, model.type = "cond", parm.type = "canon")
> theta.hat <- matrix(theta.hat, nrow = nrow(aout4$x), ncol = ncol(aout4$x))
> fit.hat <- pout4$fit
> beta.hat <- aout4$coefficients
```

We also need root data, and it will be simpler if we actually don't use the forms of the `aster` and `predict.aster` functions that take formulas (because then we don't have to cram the simulated data in a data frame and we avoid a lot of repetitive parsing of the same formulas)

```
> root <- aout4$root
> modmat <- aout4$modmat
> modmat.pred <- pout4$modmat
> x.pred <- matrix(1, nrow = dim(modmat.pred)[1], ncol = dim(modmat.pred)[2])
> root.pred <- x.pred
```

Now we're ready for a simulation

```
> set.seed(42)
> nboot <- 100
> cover <- matrix(0, nboot, length(fit.hat))
> for (iboot in 1:nboot) {
+     xstar <- raster(theta.hat, pred, fam, root)
+     aout4star <- aster(xstar, root, pred, fam, modmat, beta.hat)
+     pout4star <- predict(aout4star, x.pred, root.pred, modmat.pred,
+         amat, se.fit = TRUE)
+     upper <- pout4star$fit + crit * pout4star$se.fit
+     lower <- pout4star$fit - crit * pout4star$se.fit
+     cover[iboot, ] <- as.numeric(lower <= fit.hat & fit.hat <= upper)
+ }
> pboot <- apply(cover, 2, mean)
> pboot.se <- sqrt(pboot * (1 - pboot) / nboot)
> cbind(pboot, pboot.se)

     pboot   pboot.se
[1,]  0.91 0.02861818
[2,]  0.98 0.01400000
[3,]  0.94 0.02374868
[4,]  0.94 0.02374868
[5,]  0.91 0.02861818
[6,]  0.96 0.01959592
[7,]  0.94 0.02374868
```

Not bad for the small `nboot`. We won't try a serious simulation because it would make this tutorial run too long.

Comment Other optimization options (calling `optim` are available), but are slower. See `http://www.stat.umn.edu/geyer/trust/time.pdf`. Using `method = "nlm"` instead of the default `method = "trust"` takes 3–4 times as long, and using either method that calls `optim` (either `method = "L-BFGS-B"` or `method = "CG"`) takes about 16 times as long. So whatever problems `nlm` and `optim` is good for, they don't appear to include strictly convex objective functions like aster models have. The starting parameter value `beta.hat` is supposed to speed things up, giving the optimization a good starting point (the simulation truth). We haven't investigated whether it does or not.