Stat 8931 (Aster Models)
Lecture Slides Deck 1
Introduction and an Example

Charles J. Geyer

School of Statistics
University of Minnesota

February 19, 2020

R and License

- The version of R used to make these slides is 3.5.1.
- The version of R package `aster` used to make these slides is 1.0.2.

Aster Models

Aster models are a kind of generalized generalized linear model (that is, they generalize so-called "generalized linear models").

They are explicitly designed for **life history analysis**.

They are special cases of **regression models**, of **exponential family models**, and of **graphical models**.

Recently, aster models with **random effects** have been introduced.

Life History Analysis

Life history analysis (LHA) follows organisms over the course of their lives collecting various data: survival through various time periods and also various other data, which only makes sense conditional on survival.

Thus LHA generalizes **survival analysis**, which only uses data on survival.

The LHA of interest to my biological co-authors concerns **Darwinian fitness** conceptualized as the lifetime number of offspring an organism has. The various bits of data collected over the course of the life that contribute to this are called **components of fitness**.

Life History Analysis (cont.)

The fundamental statistical problem of LHA is that overall fitness, considered as a random variable, fits (in the statistical sense) no brand-name distribution. It has a large atom at zero (individuals that died without producing offspring) as well as multiple modes (one for each breeding season the organism survives). No statistical methodology before aster deals with data like that.

This issue has long been well understood in the LHA literature. So what was done instead was analyze components of fitness separately conditional on survival, but this doesn't address the variable (overall fitness) of primary interest (an issue also well understood, but you do what you can do).

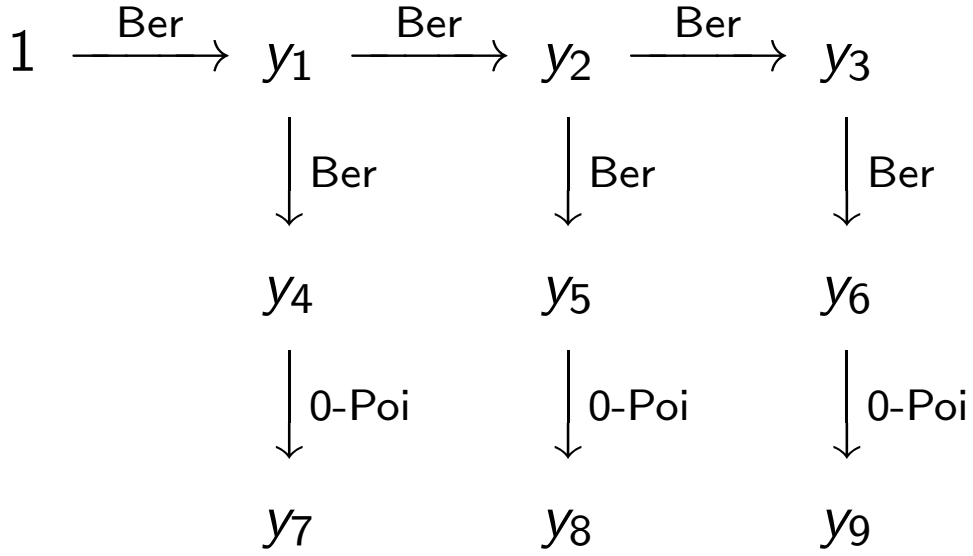
Life History Analysis (cont.)

Also there is all the data on components of fitness. That is of scientific interest too.

Aster models solve all the problems by modeling all the components of fitness jointly.

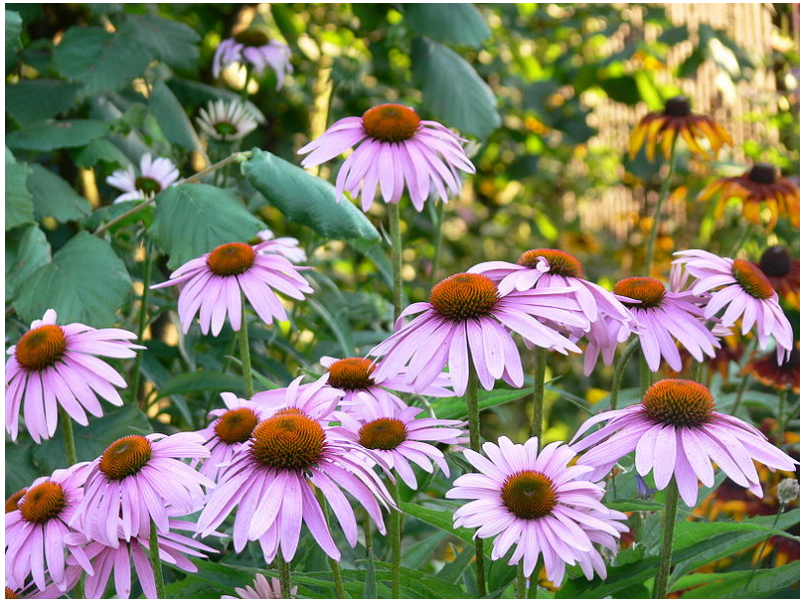
Example One

The first published example (Geyer, et al., 2007, *Biometrika*) had components of fitness for each individual in the data connected by this graph



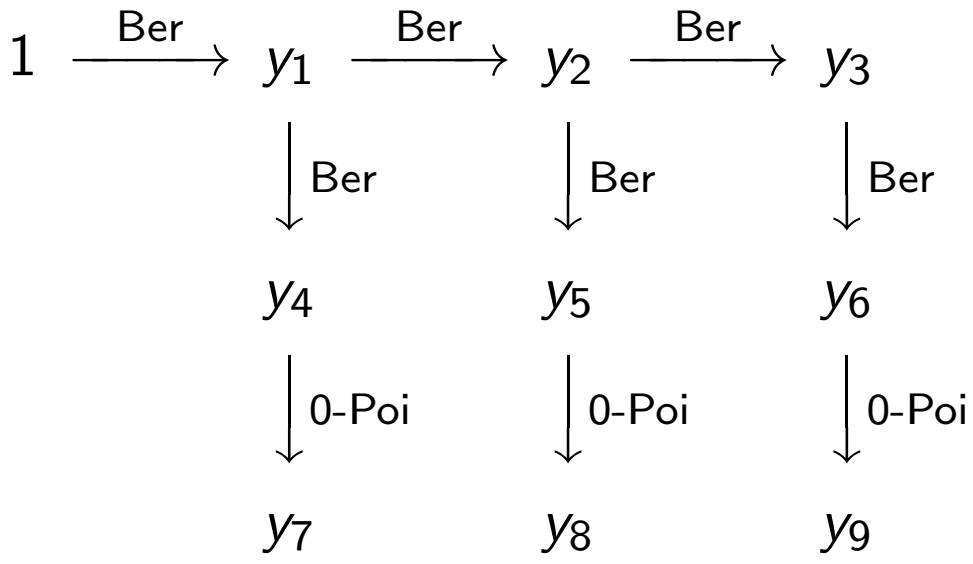
Example One (cont.)

The individuals are plants having scientific name *Echinacea angustifolia* and common name narrow-leaved purple coneflower.



They are native to the middle of North America from from Saskatchewan and Manitoba in the north to New Mexico, Texas, and Louisiana in the south.

Example One (cont.)



The y_i are the components of fitness

- y_1, y_2, y_3 are survival indicators (zero-or-one-valued) indicating survival in each of three years.
- y_4, y_5, y_6 are flowering indicators (zero-or-one-valued) indicating presence of any flowers in corresponding years.
- y_7, y_8, y_9 are flower counts in corresponding years.

A Technical Quibble

What I am calling flowers aren't. This

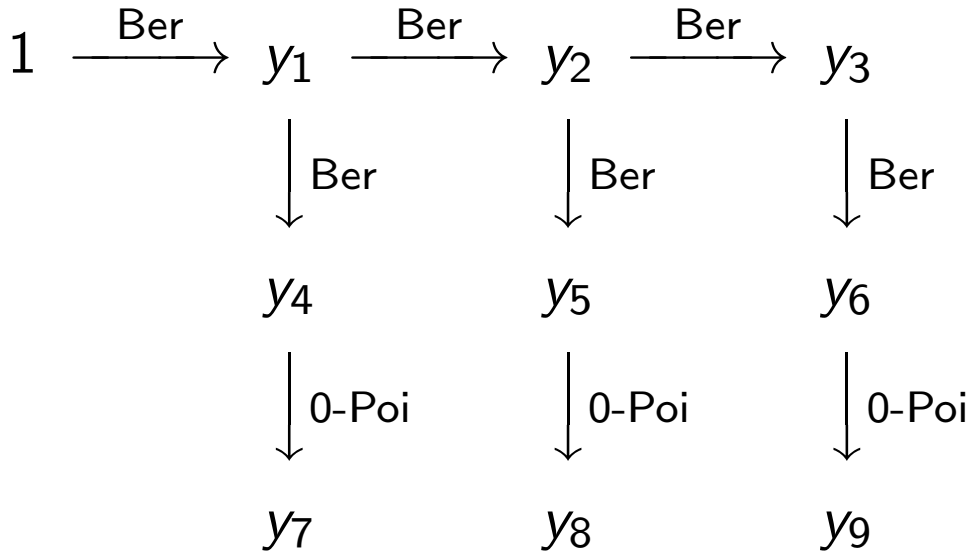
is not a flower. It is a *compound flower* or *composite flower* or *flower head*. In the picture the brown part comprises many separate flowers (also called *florets*).

The Name of the Game

These plants are in the aster family (scientific name *Asteraceae*) which used to be called *Compositae* (the family characterized by composite flowers), which is a very large family including artichoke, aster, chrysanthemum, dahlia, dandelion, daisy, goldenrod, lettuce, marigold, ragweed, safflower, sunflower, and zinnia.

Aster models are named after these flowers.

Example One (cont.)



The arrows indicate conditional distributions (Ber = Bernoulli, 0-Poi = zero-truncated Poisson).

The symbol 1 indicates the constant random variable always equal to one. A conditional distribution conditional on a constant is the same as an unconditional distribution.

Graphical Terminology

For one arrow in a graph

$$y_2 \xrightarrow{\text{Ber}} y_3$$

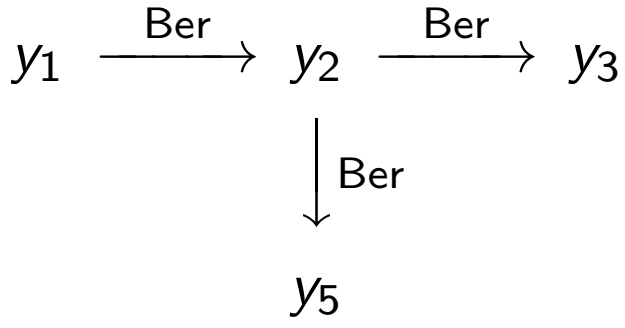
we say y_3 is the *successor* of y_2 and (conversely) we say y_2 is the *predecessor* of y_3.

A node of the graph (random variable) having no successors is called a *terminal node* of the graph.

A node of the graph (random variable) having no predecessors is called an *initial node* of the graph.

Graphical Terminology (cont.)

A node (random variable) can be both a successor and predecessor.



Here y_2 is the successor of y_1 and the predecessor of y_3 and y_5.

A terminal node can also be a successor (but not a predecessor).

An initial node can also be a predecessor (but not a successor).

Graphical Terminology (cont.)

Graph theory also uses terminology of biological origin

- child = successor
- parent = predecessor
- leaf = terminal
- root = initial

but we avoid that to avoid confusion in biological applications.

Except the R package `aster` inconsistently uses "root" instead of "initial" while using the other three non-biological terms.

We are trying to switch over to consistent terminology.

Graphical Terminology (cont.)

Graph theory terminology of biological origin has some terms that have no analog in the other terminology

- descendant = successor or successor of successor, or successor of successor of successor, and so forth
- ancestor = predecessor or predecessor of predecessor, or predecessor of predecessor of predecessor, and so forth

Fortunately, we don't need these concepts much. When we do need them, we avoid the terminology of biological origin and use the long-winded definitions instead.

Tree Graphs and Forest Graphs

Aster graphs have the property that **every node has at most one predecessor** (initial nodes have none, other nodes have one).

Aster graphs have the property that they are **acyclic** (no node is its own descendant or its own ancestor, using terminology on the preceding slide that we said we would avoid and will after this).

A graph is **connected** if every node is connected to every other node by a sequence of arrows (traversing the arrows either way).

A graph that is acyclic and in which every node has at most one predecessor is called a **forest**.

A connected forest graph is called a **tree**.

But we won't need this terminology of biological origin either. We can just say all aster models have the first two properties in boldface above.

Predecessor is Sample Size

All aster models have the **predecessor is sample size** property.

This is peculiar to aster models. No other graphical model theory has used this property.

In the subgraph

$$y_i \longrightarrow y_j$$

y_j is the sum of y_i independent and identically distributed (IID) random variables having the distribution named by the arrow label.

By convention, a sum with zero terms is zero. So $y_i = 0$ implies $y_j = 0$ with (conditional) probability one.

Predecessor is Sample Size (cont.)

$$y_i \longrightarrow y_j$$

This takes care of what people formerly conceived of as a missing data problem: when $y_i = 0$ (for concreteness, say this means the individual is dead), then you cannot "observe" y_j.

Nevertheless we can infer $y_j = 0$ (if y_j is flower count, then we are inferring that dead plants have no flowers).

So that is not a true missing data problem from the aster model perspective. Researchers do need to be aware of the need to code their data this way (dead individuals have 0 flowers not `NA` flowers, `NA` being the R value for missing data).

Predecessor is Sample Size (cont.)

If we did have truly missing data (not observable and not inferrable), then we would have a problem that aster models are not equipped to solve.

Predecessor is Sample Size (cont.)

$$1 \xrightarrow{\text{Ber}} y_1 \xrightarrow{\text{Ber}} y_2$$

Here the unconditional distribution of y_1 is Bernoulli (binomial with sample size one, the only possible distribution of a zero-or-one-valued random variable).

The conditional distribution of y_2 given y_1 is

- degenerate, concentrated at zero if $y_1 = 0$
- Bernoulli if $y_1 = 1$

Predecessor is Sample Size (cont.)

$$1 \xrightarrow{\text{Ber}} y_1 \xrightarrow{\text{Whatever}} y_2$$

Again the unconditional distribution of y_1 is Bernoulli, hence zero-or-one-valued.

The conditional distribution of y_2 given y_1 is

- degenerate, concentrated at zero if $y_1 = 0$
- Whatever if $y_1 = 1$

We see that in the zero-or-one-valued predecessor case, the interpretation is simple. Predecessor $= 0$ implies successor $= 0$. Otherwise, the conditional distribution of the successor is the named distribution.

Predecessor is Sample Size (cont.)

But predecessors do not have to be zero-or-one-valued.

$$1 \xrightarrow{\text{Poi}} y_1 \xrightarrow{\text{Ber}} y_2$$

Now y_1 is nonnegative-integer-valued (Poi = Poisson).

Recall that the predecessor is sample size property says that the conditional distribution of y_2 given y_1 is the sum of y_1 IID random variables having the named distribution (in this case, Bernoulli).

The sum of n IID Bernoulli random variables is binomial with sample size n.

The conditional distribution of y_2 given y_1 is

- degenerate, concentrated at zero if $y_1 = 0$
- Binomial with sample size y_1 if $y_1 > 0$

Predecessor is Sample Size (cont.)

$$1 \xrightarrow{\text{Poi}} y_1 \xrightarrow{\text{Ber}} y_2$$

If the conditional distribution of y_2 given $y_1 = n$ such that $n > 0$ is binomial with sample size n rather than Bernoulli, why don't we have the arrow label say that (especially since more users have heard of binomial than Bernoulli)?

It wouldn't work in general.

$$1 \xrightarrow{\text{Poi}} y_1 \xrightarrow{\text{0-Poi}} y_2$$

The conditional distribution of y_2 given $y_1 = n$ such that $n > 0$ is the sum of n IID zero-truncated Poisson random variables, but that is not a brand-name distribution.

Predecessor is Sample Size (cont.)

What is zero-truncated Poisson anyway? And why do we want it?

Zero-truncated Poisson is a Poisson random variable conditioned on not being zero. The probability mass function (PMF) is

$$f(x) = \frac{\mu^x e^{-\mu}}{x!(1 - e^{-\mu})}, \qquad x = 1, 2, \ldots,$$

where $\mu > 0$ is the mean of the untruncated Poisson variable, (just the Poisson PMF divided by the probability the Poisson variable is nonzero, which is $1 - e^{-\mu}$).

Predecessor is Sample Size (cont.)

The reason why we want it is that sometimes random variables are zero for reasons other than Poisson variation.

If the variable is flower count, then sometimes there are no flowers for reasons other than Poisson variation (maybe deer ate them all).

If we want to deal with this we need the so-called zero-inflated Poisson distribution (about which there has been much recent literature, about 13,000 hits in Google Scholar).

Predecessor is Sample Size (cont.)

For reasons that will be discussed later the aster model way to get the zero-inflated Poisson distribution uses two arrows rather than one

$$y_i \xrightarrow{\text{Ber}} y_j \xrightarrow{\text{0-Poi}} y_k$$

The conditional distribution of y_k given y_i (both arrows) is

- degenerate, concentrated at zero if $y_i = 0$
- zero-inflated Poisson, if $y_i = 1$
- the sum of y_i IID zero-inflated Poisson random variables, if $y_i > 1$

A Technical Quibble

$$y_i \xrightarrow{\text{Ber}} y_j \xrightarrow{\text{0-Poi}} y_k$$

Strictly speaking, the conditional distribution of y_k given $y_i = 1$ is zero-inflated-or-deflated Poisson because it imposes no constraint that

$$\Pr(y_k = 0 \mid y_i = 1)$$

be greater than that under the Poisson distribution.

We will ignore this quibble because it is of little scientific interest.

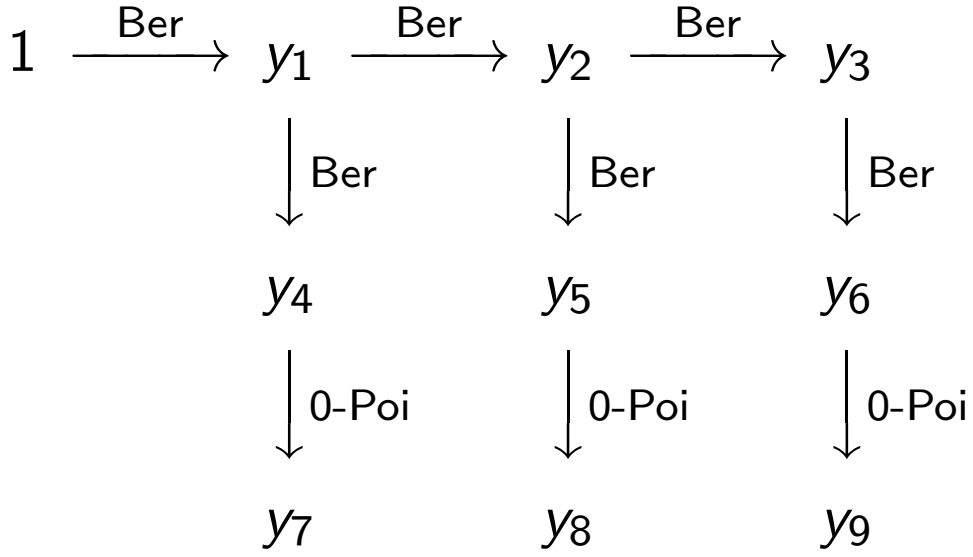
Caution

The zero-truncated Poisson distribution is for when the zero-truncated successor is zero **if and only if** its predecessor is zero.

Usually, this happens when the predecessor is **defined** to be zero if and only if the zero-truncated successor is zero.

If you are not sure whether you should use Poisson or zero-truncated Poisson, the answer is usually Poisson.

Example One (cont.)



Now we can understand the graph for the example.

The top layer (survival indicators) are necessary to model survival components of fitness.

The middle and bottom layers (flowering indicators and flower counts) are necessary to model fecundity components of fitness while accounting for zero-inflation of the Poisson distributions.

Example One (cont.)

Data for example one are found in the R dataset `echinacea` in the R package aster.

```
> library(aster)
> data(echinacea)
> class(echinacea)

[1] "data.frame"

> dim(echinacea)

[1] 570  12

> names(echinacea)

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

R Data Import

How would you get your own data into R to make a similar data frame?

Many, many different ways. R can read many data formats. There is a whole book about it on the web.

R Data Import/Export
`http://cran.r-project.org/doc/manuals/r-release/R-data.html`

We will just explain two ways.

R Data Import (cont.)

Put the data in a plain text file, data for a single individual in each row, white-space-separated columns (if you have white space inside individual items, they must be quoted strings like `"foo bar"`), variable names are column headings in the first line of the file.

For example, the plain text file that R uses to read in the `echinacea` dataset starts

```
"hdct02" "hdct03" "hdct04" "pop" "ewloc" "nsloc" "ld02" "fl
0 0  0 "NWLF"  -8 -11 0 0 0 0 0 0
0 0  0 "Eriley"  -8 -10 1 0 1 0 1 0
0 0  0 "NWLF"  -8  -9 0 0 0 0 0 0
0 0  0 "SPP"  -8  -8 0 0 0 0 0 0
```

(the first line runs off the screen but has all the variable names).

R Data Import (cont.)

If you want to get a copy of the whole file to look at

```
library(aster)
data(echinacea)
write.table(echinacea, file = "foo.txt",
    row.names = FALSE)
```

does that. Then

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

does the reverse operation, reading the table back and assigning it the name `foo`.

R Data Import (cont.)

If you have put the data in Microsoft Excel or LibreOffice Calc, write out the data as a CSV (comma separated values) file, then read it with

```
foo <- read.csv(file = "foo.csv")
```

(assuming you wrote out the data into the file `foo.csv`).

R Data Import (cont.)

Caution:

You cannot write a "plain text file" with Microsoft Word or other "word processor" program. They put in lots of additional stuff besides text. (You may get a plain text file if you choose "Text" as your output format.)

Microsoft Notepad and Notepad++ do write plain text files.

You can put all kinds of stuff in a spreadsheet, but `read.csv` wants one line per individual plus a header line (with variable names) and nothing else! Delete everything else before writing this out as a CSV file to read into R.

Example One (cont.)

The variables that correspond to nodes of the graph are, in the order they are numbered in the graph

```
> vars <- c("ld02", "ld03", "ld04", "fl02", "fl03",
+     "fl04", "hdct02", "hdct03", "hdct04")
```

The graphical structure is specified by a vector that gives for each node the index (not the name) of the predecessor node or zero if the predecessor is an initial node.

```
> pred <- c(0, 1, 2, 1, 2, 3, 4, 5, 6)
```

This says the predecessor of the first node given by the `vars` vector is initial (because `pred[1] == 0`), the predecessor of the second node given by the `vars` vector is the first node given by the `vars` vector (because `pred[2] == 1`), and so forth.

Example One (cont.)

Let's check this makes sense.

```
> foo <- rbind(vars, c("initial", vars)[pred + 1])
> rownames(foo) <- c("successor", "predecessor")
> foo
```

```
             [,1]      [,2]   [,3]   [,4]   [,5]
successor   "ld02"    "ld03" "ld04" "fl02" "fl03"
predecessor "initial" "ld02" "ld03" "ld02" "ld03"
             [,6]   [,7]     [,8]     [,9]
successor   "fl04" "hdct02" "hdct03" "hdct04"
predecessor "ld04" "fl02"   "fl03"   "fl04"
```

That's right.

Example One (cont.)

The last part of the specification of the graph is given by a corresponding vector of integers coding families (distributions). The default is to use the codes: 1 = Bernoulli, 2 = Poisson, 3 = zero-truncated Poisson. Optionally, the integer codes specify families given by an optional argument `famlist` to functions in the `aster` package, and this can specify other distributions besides those in the default coding.

```
> fam <- c(1, 1, 1, 1, 1, 1, 3, 3, 3)
> rbind(vars, fam)
```

```
     [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
vars "ld02" "ld03" "ld04" "fl02" "fl03" "fl04"
fam  "1"    "1"    "1"    "1"    "1"    "1"
     [,7]     [,8]     [,9]
vars "hdct02" "hdct03" "hdct04"
fam  "3"      "3"      "3"
```

Example One (cont.)

There is one more step before we can fit models. The R function `aster` which fits aster models wants the data in "long" rather than "wide" format, the former having one line per node of the graph rather than one per individual.

The magic incantation to do this is

```
> redata <- reshape(echinacea, varying = list(vars),
+     direction = "long", timevar = "varb",
+     times = as.factor(vars), v.names = "resp")
> redata <- data.frame(redata, root = 1)
```

If you forget this incantation, it and everything else we have done in this example is on the help page for the R function `aster` obtained by doing

```
help(aster)
```

Example One (cont.)

```
> class(redata)

[1] "data.frame"

> dim(redata)

[1] 5130    7

> sapply(redata, class)

      pop     ewloc     nsloc      varb      resp
 "factor" "integer" "integer"  "factor" "integer"
       id      root
"integer" "numeric"
```

Example One (cont.)

```
> names(redata)

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

All of the variables in `echinacea` that are named in `vars` are gone. They are packed into the variable `resp`. Which components of `resp` correspond to which components of `vars` is shown by the new variable `varb`

```
> levels(redata$varb)

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

Example One (cont.)

Now we have all of the response variables (components of fitness) collected into a single vector `resp` and we have learned what `varb` is. What about the other variables?

`root` we defined ourselves. When the predecessor of a node is initial, then the corresponding component of `root` gives the value of the predecessor. Other components of `root` are ignored. We set them all to one.

`id` is seldom (if ever) used. It tells what individual (what row of the original data frame `echinacea`) a row of `reshape` came from.

`nsloc` (north-south location) and `ewloc` (east-west location) give the position each individual was located in the experimental plot.

Example One (cont.)

pop gives the ancestral populations: each individual was grown from seed taken from a surviving population in a prairie remnant in western Minnesota near the Echinacea Project field site.

```
> levels(redata$pop)

[1] "AA"       "Eriley"   "Lf"       "Nessman" "NWLF"
[6] "SPP"      "Stevens"
```

The R Formula Mini-Language

In R, regression modeling functions (and other functions) use what I call the "R formula mini-language" and everybody else (including R itself) just calls "formulas".

```
> foo <- as.formula("y ~ x")
> foo

y ~ x

> class(foo)

[1] "formula"
```

The R Formula Mini-Language (cont.)

I call it a mini-language because it is one.

The mini-language documentation (such as it is) is shown by

```
help(formula)
```

But the language isn't really well documented. The language is what its interpreter says it is. That "interpreter" is the R function `model.matrix`, which turns formulas and data frames into model matrices. Ordinary users never call this function directly. Regression modeling functions like `lm`, `glm`, and `aster` call `model.matrix` to do the job.

The R Formula Mini-Language (cont.)

An R formula "means" what `model.matrix` says it means.

If you are ever in doubt as to what a formula means, look at the model matrix produced. That is definitive. The model matrix determines the regression model.

The R Formula Mini-Language (cont.)

In the R formula mini-language the following characters are magic

- ~ twiddle (officially "tilde")
- + plus
- − minus
- : colon
- * star (officially "asterisk")
- ^ hat (officially "caret")

They do not mean in formulas what they mean elsewhere in R.

The R Formula Mini-Language (cont.)

For example, in the formula `y ~ x + z` the symbol + does not mean to add the vectors `x` and `z` as it would in a non-formula R expression.

If the variables `x` and `z` are quantitative, then the formula `y ~ x + z` means

$$\eta_i = \beta_1 + \beta_2 x_i + \beta_3 z_i$$

where the β_i are unknown parameters, x_i are the components of the vector `x` and similarly for z_i, and η_i is another parameter being specified in terms of these things.

The regression coefficients (the β_i) don't appear in the formula. There isn't a "term" in the formula corresponding to β_1. R always puts in an "intercept" term unless you ask it not to.

The R Formula Mini-Language (cont.)

So + in a formula does not mean what + means in a non-formula.

In a formula it means something like the thingummies connected by + signs ("terms") get multiplied by unknown parameters ("regression coefficients") before being added.

And this doesn't take into account more complicated issues that arise when the other magic characters are involved.

The R Formula Mini-Language (cont.)

In `y ~ x + z` meaning

$$\eta_i = \beta_1 + \beta_2 x_i + \beta_3 z_i$$

The twiddle means the vector η having components η_i has something to do with the vector `y` ("response" vector)

For linear models (fit by the R function `lm`) $\eta_i = E(y_i)$.

For generalized linear models (fit by the R function `glm`) η_i is some monotone function of $E(y_i)$ ("link" function).

For aster models (fit by the R function `aster`) the vector η having components η_i is a **multivariate monotone** function of the vector $E(y)$ having components $E(y_i)$.

The R Formula Mini-Language (cont.)

The R formula mini-language doesn't care about the interpretation of the η_i.

All it cares about is that η_i is some function (which its job is to specify) of unknown parameters ("regression coefficients") and other thingummies ("predictor" variables or "covariates").

This function is always linear in the regression coefficients. It need not be linear in the predictors.

> It's called "linear regression" because it's linear in the regression coefficients, not because it is linear in x.
>
> — Werner Stutzle

Categorical Predictors and Dummy Variables

If a covariate in a formula is categorical (the R object type for this is `factor` — examples are `varb` and `pop` in our example data), then the R formula mini-language interpreter (`model.matrix`) turns them into a set of zero-or-one-valued variables ("indicator" variables, also called "dummy" variables).

Non-numeric variables (such as character variables) are automatically turned into factors when stuffed into a data frame by `read.table` or `read.csv` (unless the optional argument `stringsAsFactors = FALSE` is supplied). If you want a numeric variable to be treated as a factor, you have to explicitly say so

```
redata <- transform(redata, fred = as.factor(fred))
```

Categorical Predictors and Dummy Variables (cont.)

If we have a factor `color` with possible values (R calls them "levels") `"red"`, `"blue"`, and `"green"` and a quantitative variable `x`, then

```
y ~ color + x
```

means

$$\eta_i = \beta_1 + \beta_2 u_i + \beta_3 v_i + \beta_4 x_i$$

where u_i and v_i are components of zero-or-one-valued vectors u and v created by `model.matrix`, u indicating green individuals and v indicating red individuals.

Regression Models

Now we are ready to fit some aster models to our example data.

But we still have to explain regression models in the aster context.

And really understanding that involves all of the theory of aster models, most of which we have skipped, wanting to get to a real concrete example.

So what we need to do next, we are not ready for.

So in order to get on to the example, we are going to have to oversimplify (dumb down to the point of actually being wrong).

Everything until further notice should be taken *cum grano salis*.

Regression Models (cont.)

In ordinary regression models, components of the response vector are

- independent and
- in the same family

In aster models, components of the response vector are

- dependent (the dependence being specified by the graph) and
- in different families

Regression Models (cont.)

Different families for different nodes of the graph means it makes no sense to have terms of the regression formula applying to different nodes.

In particular, it makes no sense to have one "intercept" for all nodes.

To in effect get a different "intercept" for each node in the graph, include `varb` in the formula

```
y ~ varb + ...
```

The categorical variable `varb` gets turned into as many dummy variables as there are nodes in the graph, one is dropped, and the "intercept" dummy variable (all components $= 1$) is added; the effect is to provide a different intercept for each node.

A Technical Quibble

Why is does the variable named `varb` have that name?

Because of the optional argument `timevar = "varb"` supplied to the "magic incantation" (`reshape` function)

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

We could have given it the name `fred` or `sally` or whatever we want. I picked `varb` (short for "variables") without thinking about it the first time I did this, and everyone (including me) has just copied that ever since.

Similarly the name `resp` for the response is specified by the optional argument `v.names = "resp"`.

Regression Models (cont.)

Similar thinking says we want completely different regression coefficients of all kinds of predictors for each node of the graph. That would lead us to formulas like

```
y ~ varb + varb : (...)
```

where ... is any other part of the formula.

The `:` operator in the R formula mini-language denotes interactions without main effects. The `*` operator in the R formula mini-language denotes interactions with main effects. That is, `a * b` means the same thing as `a + b + a : b`

So the above formula says we want the "main effects" for `varb`, and we want the "interaction" of `varb` with "everything else" (the ...), but we do not want the "main effects" for "everything else".

Regression Models (cont.)

```
y ~ varb + varb : (...)
```

Having said that, we immediately want to take it back. The language of "main effects" and "interactions" was never designed to apply to aster models.

We should not think of this formula as specifying "main effects" for `varb` (whatever that may mean) but rather as specifying a separate "intercept" for each node of the graph.

Similarly, we should not think of this formula as specifying "interaction" between `varb` and "everything else" (whatever that may mean) but rather as specifying separate coefficients for "everything else" for each node of the graph.

Regression Models (cont.)

Thus IMHO (in my humble opinion) you should always say "main effects" and "interactions" with scare quotes, emphasizing that these terms are at best highly misleading and confusing.

Regression Models (cont.)

```
y ~ varb + varb : (...)
```

But formulas like this would yield too many regression coefficients to estimate well! We can do better!

Maybe we don't really need different regression coefficients for each node. Maybe different for each kind of node (whatever that may mean) would be enough.

Regression Models (cont.)

```
> layer <- gsub("[0-9]", "", as.character(redata$varb))
> unique(layer)

[1] "ld"   "fl"   "hdct"

> redata <- data.frame(redata, layer = layer)
> with(redata, class(layer))

[1] "factor"
```

Maybe

```
y ~ varb + layer : (...)
```

good enough?

Regression Models (cont.)

```
y ~ varb + layer : (...)
```

But formulas like this would still yield too many regression coefficients to estimate well! We can do better!

In aster models (and this is really where the explanation gets dumbed down to the point of being wrong) regression coefficients "for" a node of the graph also influence all "earlier" nodes of the graph (predecessor, predecessor of predecessor, predecessor of predecessor of predecessor, etc.)

So maybe it would be good enough to only have separate coefficients for the "layer" of the graph consisting of terminal nodes?

Regression Models (cont.)

```
> fit <- as.numeric(layer == "hdct")
> unique(fit)

[1] 0 1

> redata <- data.frame(redata, fit = fit)
> with(redata, class(fit))

[1] "numeric"
```

Maybe

```
y ~ varb + fit : (...)
```

good enough?

Regression Models (cont.)

We called this variable we just made up `fit`, short for Darwinian fitness.

With formulas like

```
y ~ varb + fit : (...)
```

the regression coefficients in terms specified by ... have a direct relationship with expected Darwinian fitness. And that's usually what is wanted in LHA.

A Technical Quibble

We shouldn't have said Darwinian fitness. Rather we should have said *the best surrogate of Darwinian fitness in these data*.

Flower counts are not "lifetime number of offspring". Still less are flower counts over three years (not the whole life span).

Other *Echinacea* data (Wagenius, et al., 2010, *Evolution*) have more years and more components of fitness.

Other data on other species (Stanton-Geddes, et al., 2012, *PLoS One*) have "best surrogate of fitness" pretty close to "fitness" (with no qualifiers).

After we have emitted academic weasel-wording making clear that we are aware of the difference between what we are calling fitness and the Platonic ideal of fitness (Should we be essentialists about fitness? Isn't that an oxymoron?) we can just drop the fuss and go on with the analysis and interpretation.

Regression Models (cont.)

In practice we use formulas like

```
y ~ varb + layer : (...) + fit : (...)
```

with the two ... having different formula terms.

The formula terms in the second ... are the ones that we want to say have a direct effect on fitness (and want statistics to tell us whether they do or not).

The formula terms in the first ... are everything else (the terms whose effect on fitness, if any, is not an issue of scientific interest in this experiment).

No Naked Predictors

We summarize our advice about formulas for aster models with the slogan

> No naked predictors!

or more precisely

> No naked predictors except `varb` and factor (indicator) variables derived from it, like `layer` and `fit`

Our slogan means every predictor other than these must occur "interacted with" one of these.

No Naked Predictors (cont.)

Instead of

```
y ~ varb + layer : (...) + fit : (...)
```

why not

```
y ~ varb + layer + fit + layer : (...) + fit : (...)
```

Because the dummy variables constructed from `layer` and the dummy variable which is `fit` are linear combinations of the dummy variables constructed from `varb` and so would be dropped anyway. (The two formulas specify the same model.)

Regression Models (cont.)

And that ends the *cum grano salis* explanation of regression models. We return to telling it like it is.

Example One (cont.)

```
> aout <- aster(resp ~ varb + layer : (nsloc + ewloc) +
+     fit : pop, pred, fam, varb, id, root, data = redata)
> summary(aout)

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

                Estimate Std. Error z value Pr(>|z|)
(Intercept)    -1.050644   0.184332  -5.700 1.20e-08
varbfl03       -0.349096   0.267919  -1.303   0.1926
varbfl04       -0.344222   0.243899  -1.411   0.1581
varbhdct02      1.321414   0.261174   5.060 4.20e-07
varbhdct03      1.343374   0.214625   6.259 3.87e-10
varbhdct04      1.851328   0.199853   9.263  < 2e-16
varbld02       -0.029302   0.315703  -0.093   0.9260
varbld03        1.740051   0.396189   4.392 1.12e-05
```

Example One (cont.)

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-1.0506435	0.1843320	-5.6997	1.200e-08
varbfl03	-0.3490958	0.2679185	-1.3030	0.19258
varbfl04	-0.3442222	0.2438992	-1.4113	0.15815
varbhdct02	1.3214136	0.2611741	5.0595	4.203e-07
varbhdct03	1.3433740	0.2146250	6.2592	3.870e-10
varbhdct04	1.8513276	0.1998528	9.2635	< 2.2e-16
varbld02	-0.0293022	0.3157033	-0.0928	0.92605
varbld03	1.7400507	0.3961890	4.3920	1.123e-05
varbld04	4.1885771	0.3342661	12.5307	< 2.2e-16
layerfl:nsloc	0.0701024	0.0146520	4.7845	1.714e-06
layerhdct:nsloc	-0.0058043	0.0055499	-1.0458	0.29564
layerld:nsloc	0.0071652	0.0058667	1.2213	0.22196
layerfl:ewloc	0.0179769	0.0144128	1.2473	0.21229
layerhdct:ewloc	0.0076060	0.0055608	1.3678	0.17138
layerld:ewloc	-0.0047874	0.0059191	-0.8088	0.41863
fit:popAA	0.1292377	0.0891292	1.4500	0.14706
fit:popEriley	-0.0495612	0.0712789	-0.6953	0.48686
fit:popLf	-0.0332786	0.0795727	-0.4182	0.67579
fit:popNessman	-0.1862690	0.1277869	-1.4577	0.14494
fit:popNWLF	0.0210283	0.0635998	0.3306	0.74092
fit:popSPP	0.1491795	0.0677156	2.2030	0.02759

Example One (cont.)

The regression coefficients are of little interest. The main interest is in what model among those that have a scientific interpretation fits the best.

```
> aout.smaller <- aster(resp ~ varb +
+     fit : (nsloc + ewloc + pop),
+     pred, fam, varb, id, root, data = redata)
> aout.bigger <- aster(resp ~ varb +
+     layer : (nsloc + ewloc + pop),
+     pred, fam, varb, id, root, data = redata)
```

Example One (cont.)

```
> anova(aout.smaller, aout, aout.bigger)

Analysis of Deviance Table

Model 1: resp ~ varb + fit:(nsloc + ewloc + pop)
Model 2: resp ~ varb + layer:(nsloc + ewloc) + fit:pop
Model 3: resp ~ varb + layer:(nsloc + ewloc + pop)
  Model Df Model Dev Df Deviance P(>|Chi|)
1       17   -2746.7
2       21   -2712.5  4   34.203 6.772e-07 ***
3       33   -2674.7 12   37.838 0.0001632 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Example One (cont.)

Despite the largest model fitting the best, we choose the middle model because that one tells us something about fitness directly that the other one does not.

We haven't covered enough aster model theory to explain this. More on this later (end of Deck 4 of these slides).

Predicted Values

Now we come to a really hard subject for applied work. Hypothesis tests using the R function `anova` are fairly straightforward. Confidence intervals using the R function `predict` are anything but straightforward (crooked as a dog's hind leg).

Part of this is bad design. The `predict` function has some aspects of its user interface that are clumsy and hard to use, even for the author of the function. Unfortunately, they cannot be fixed without breaking a lot of working examples (which would be much worse than just living with these issues). The R package `aster2` fixes these issues (and does lots more) but is still incomplete.

Predicted Values (cont.)

But the other part of what makes confidence intervals is just the inherent complexity of aster models.

Whatever you personally are trying to do with aster models is a very special case of what aster models can do. As we shall see, they can do many things that look radically different and have no obvious connection with each other. (They don't have any obvious similarities in their data or scientific interpretations of their data. The only connection is aster model theory applies to all of them.)

Among other issues, aster models have six (!) different parameterizations, *all of which can be of scientific interest in some application*, not necessarily in your application, not necessarily all in any one application.

R Generic Functions

R has a feature called **generic functions**.

A generic function, like R functions `print`, `plot`, `summary`, and many more do different things for different objects.

What they do depends on the **class** of the object, which is what the R function `class` tells us.

```
> class(aout)
```

```
[1] "aster.formula"  "aster"              "asterOrReaster"
```

Here we see an object can be in multiple classes.

R Generic Functions (cont.)

Given a class name, we can find out what R generic functions have special methods for it.

```
> foo <- lapply(class(aout),
+     function(x) methods(class = x))
> foo <- lapply(foo, as.vector)
> foo <- unlist(foo)
> foo <- sort(unique(foo))
> foo
```

```
[1] "anova.asterOrReaster"  "predict.aster"
[3] "predict.aster.formula" "summary.aster"
```

R Generic Functions (cont.)

In these, the first part (before the class name) is the generic function name. We see that R package aster provides methods for three R generic functions.

```
> unique(sub("\\..*$", "", foo))
```

```
[1] "anova"   "predict" "summary"
```

R Generic Functions (cont.)

When invoking a generic function one normally only uses the generic function name (not the method name, the long names we saw above).

For example

```
> moo <- predict(aout)
```

The method `predict.aster.formula` is automatically called because `aout` has class `"aster.formula"`.

R Generic Functions (cont.)

And the main reason users have to know any of this is that in order to see the documentation for this function you have to say

```
help(predict.aster.formula)
```

or

```
help(predict.aster)
```

(both of these happen to take you to the same help page). Just saying

```
help(predict)
```

will not show you what you want.

Predicted Values (cont.)

```
> pout <- predict(aout)
> class(pout)

[1] "numeric"

> length(pout)

[1] 5130

> nrow(redata)

[1] 5130
```

Predicted Values (cont.)

`predict.aster` and `predict.aster.formula` have many complicated options. When invoked with no optional arguments (as just shown), it produces a numeric vector of the same length as the response vector.

The result of `predict(aout)` is the maximum likelihood estimate (MLE) of the *saturated model mean value parameter vector* μ.

If y denotes the response vector, then

$$E(y) = \mu$$

meaning

$$E(y_i) = \mu_i$$

(the components of μ are the unconditional expectations of the corresponding components of y).

Predicted Values (cont.)

As everywhere else in statistics we distinguish parameters like μ from their estimates $\hat{\mu}$. We say μ is the unknown true parameter (vector) value that determined the distribution of the data, and $\hat{\mu}$ is only an estimator of μ.

If we want to say how bad or good our estimators are, then we need confidence intervals (or perhaps just standard errors).

```
> pout <- predict(aout, se.fit = TRUE)
> class(pout)

[1] "list"

> sapply(pout, class)

      fit    se.fit  gradient
"numeric" "numeric"  "matrix"
```

Predicted Values (cont.)

- The component `fit` gives the estimators (the same vector that was returned when `predict` was invoked with no optional arguments).
- The component `se.fit` gives the corresponding standard errors.
- The component `gradient` gives the derivative of the map from regression coefficients to predictions.

These are asymptotic (large sample size, approximate) estimated standard deviations of the components of $\hat{\mu}$ derived using the "usual" theory of maximum likelihood estimation (more on that later).

Predicted Values (cont.)

```
> low <- pout$fit - qnorm(0.975) * pout$se.fit
> hig <- pout$fit + qnorm(0.975) * pout$se.fit
> length(hig)
```

```
[1] 5130
```

gives us vector containing confidence bounds for approximate 95% confidence intervals (*not corrected for simultaneous coverage!*) for each of the components of the response vector.

These are of no scientific interest whatsoever.

Predicted Values (cont.)

The question of scientific interest addressed by confidence intervals in the first aster paper was about (best surrogate of) fitness of a *typical* individual in each population. Thus we only want

```
> nlevels(redata$pop)
```

```
[1] 7
```

confidence intervals, one for each population.

What do we mean by "typical" individuals? Those that are directly comparable. Those that the same in all respects except for population.

In particular, they should be planted at exactly the same place (have the same values of `nsloc` and `ewloc`). Clearly, real individuals are not comparable in this way. (Two different plants cannot have the same location.)

Predicted Values (cont.)

Thus we have to *make up covariate data* for *hypothetical* individuals that are comparable like this and get estimated mean values for them.

```
> fred <- data.frame(nsloc = 0, ewloc = 0,
+     pop = levels(redata$pop), root = 1,
+     ld02 = 1, ld03 = 1, ld04 = 1,
+     fl02 = 1, fl03 = 1, fl04 = 1,
+     hdct02 = 1, hdct03 = 1, hdct04 = 1)
> fred
```

	nsloc	ewloc	pop	root	ld02	ld03	ld04	fl02	fl03
1	0	0	AA	1	1	1	1	1	1
2	0	0	Eriley	1	1	1	1	1	1
3	0	0	Lf	1	1	1	1	1	1
4	0	0	Nessman	1	1	1	1	1	1
5	0	0	NWLF	1	1	1	1	1	1
6	0	0	SPP	1	1	1	1	1	1
7	0	0	Stevens	1	1	1	1	1	1

Predicted Values (cont.)

Seems to work. The components of the response vector are ignored in prediction so we can give them arbitrary values. Somewhat annoyingly, they have to be possible values because `predict.aster.formula` will check.

```
> renewdata <- reshape(fred, varying = list(vars),
+     direction = "long", timevar = "varb",
+     times = as.factor(vars), v.names = "resp")
> layer <- gsub("[0-9]", "", as.character(renewdata$varb))
> renewdata <- data.frame(renewdata, layer = layer)
> fit <- as.numeric(layer == "hdct")
> renewdata <- data.frame(renewdata, fit = fit)
```

We did exactly the same things we did to make `redata` in making `renewdata` changing what had to be changed (*mutatis mutandis* as the economists say).

Predicted Values (cont.)

Now we have predictions for these guys

```
> names(renewdata)

[1] "nsloc" "ewloc" "pop"   "root"  "varb"  "resp"
[7] "id"    "layer" "fit"

> pout <- predict(aout, newdata = renewdata, varvar = varb,
+     idvar = id, root = root, se.fit = TRUE)
> sapply(pout, class)

      fit    se.fit  gradient    modmat
"numeric" "numeric"  "matrix"   "array"

> sapply(pout, length)

     fit   se.fit gradient   modmat
      63       63     1323     1323
```

Predicted Values (cont.)

Why do we need the arguments `varvar`, `idvar`, and `root` when we didn't before? Dunno. More bad design. But `help(predict.aster)` says we need them (especially look at the examples, which is always good advice).

So now we can make 63 not corrected for simultaneous coverage confidence intervals, one for each of the 9 nodes of the graph for each of these 7 individuals (one per population).

These too are of no scientific interest whatsoever. But we are getting closer.

Predicted Values (cont.)

What is of scientific interest is confidence intervals for Darwinian fitness for these 7 individuals.

Fitness (best surrogate of) in these data is the lifetime headcount which is

$$\texttt{hdct02} + \texttt{hdct03} + \texttt{hdct04}$$

where the variable names here are meant to indicate the actual variables.

What? Wait!

$$\texttt{hdct02} + \texttt{hdct03} + \texttt{hdct04}$$

is fitness? What about the other components of fitness? Don't they contribute too?

Yes, they do. But their effect is already counted in the head count. You can't have nonzero head count if you are dead or if you had no flowers, so that is already accounted for.

What? Wait! (cont.)

We distinguish between *observed (Darwinian) fitness* and *expected (Darwinian) fitness*.

Observed fitness is a random variable, ideally the lifetime number of offspring an individual has. This is highly variable and mostly the result of environmental accident rather than underlying genetics.

Expected fitness is a constant (not a random variable — *every expectation* is a constant not a random variable). Expected fitness is the (unconditional) expectation of observed fitness.

Expected fitness is what we are trying to do confidence intervals for here.

What? Wait! (cont.)

When we say the effect of other components of fitness (`ld0`x and `fl0`x is already counted in head count (`hdct0`x), what we mean is that the **unconditional expectation** of head count incorporates the effect of these earlier components of fitness.

This will become obvious when we do the theory in the next deck of slides.

Predicted Values (cont.)

Getting the predicted values is no problem if we know the order the nodes of the graph are arranged in, which is shown by

```
> renewdata$id
```

```
 [1] 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4
[26] 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1
[51] 2 3 4 5 6 7 1 2 3 4 5 6 7
```

```
> as.character(renewdata$varb)
```

```
 [1] "ld02"   "ld02"   "ld02"   "ld02"   "ld02"
 [6] "ld02"   "ld02"   "ld03"   "ld03"   "ld03"
[11] "ld03"   "ld03"   "ld03"   "ld03"   "ld04"
[16] "ld04"   "ld04"   "ld04"   "ld04"   "ld04"
[21] "ld04"   "f102"   "f102"   "f102"   "f102"
[26] "f102"   "f102"   "f102"   "f103"   "f103"
[31] "f103"   "f103"   "f103"   "f103"   "f103"
[36] "f104"   "f104"   "f104"   "f104"   "f104"
[41] "f104"   "f104"   "hdct02" "hdct02" "hdct02"
```

Predicted Values (cont.)

We see it runs through all individuals for each node before going on to the next node. So

```
> nnode <- length(vars)
> sally <- matrix(pout$fit, ncol = nnode)
> dim(sally)
```

```
[1] 7 9
```

```
> rownames(sally) <- unique(as.character(renewdata$pop))
> colnames(sally) <- unique(as.character(renewdata$varb))
```

stuffs the parameter estimates into a matrix with individuals along rows and nodes along columns.

By default R uses FORTRAN order when stuffing a vector into a matrix, filling a column before going on to the next (this can be changed by the optional argument `byrow = TRUE`).

Predicted Values (cont.)

```
> round(sally, 4)

           ld02   ld03   ld04   fl02   fl03   fl04
AA       0.7834 0.7521 0.7285 0.3229 0.2560 0.4561
Eriley   0.6954 0.6565 0.6299 0.2334 0.1774 0.3237
Lf       0.7029 0.6646 0.6382 0.2404 0.1834 0.3342
Nessman  0.6377 0.5946 0.5669 0.1824 0.1348 0.2469
NWLF     0.7289 0.6926 0.6670 0.2655 0.2051 0.3716
SPP      0.7937 0.7634 0.7402 0.3346 0.2666 0.4729
Stevens  0.7187 0.6816 0.6557 0.2555 0.1964 0.3567
         hdct02 hdct03 hdct04
AA       0.6215 0.4990 1.2555
Eriley   0.4085 0.3140 0.7796
Lf       0.4242 0.3273 0.8144
Nessman  0.2993 0.2233 0.5418
NWLF     0.4817 0.3764 0.9422
SPP      0.6514 0.5257 1.3224
Stevens  0.4585 0.3565 0.8905
```

Predicted Values (cont.)

```
> herman <- sally[ , grepl("hdct", colnames(sally))]
> herman
```

```
           hdct02    hdct03    hdct04
AA       0.6215239 0.4990070 1.2554533
Eriley   0.4084934 0.3139651 0.7796097
Lf       0.4242352 0.3272954 0.8144317
Nessman  0.2993480 0.2233300 0.5418002
NWLF     0.4816654 0.3764236 0.9421609
SPP      0.6513874 0.5256736 1.3224073
Stevens  0.4584965 0.3565129 0.8905197
```

```
> rowSums(herman)
```

```
      AA   Eriley        Lf  Nessman     NWLF      SPP
2.375984 1.502068 1.565962 1.064478 1.800250 2.499468
 Stevens
1.705529
```

Predicted Values (cont.)

These are the desired estimates of expected fitness, but they don't come with standard errors because there is no simple way to get the standard errors for sums from the standard errors for the summands (when the summands are not independent, which is the case here).

So we have to proceed indirectly. We have to tell `predict.aster.formula` what functions of mean values we want and let it figure out the standard errors (which it can do).

It only figures out for *linear functions*. We can handle non-linear functions using the delta method "by hand" (using R as a calculator but doing derivatives ourselves), but that is much more complicated. Since addition is a linear operation, we do not need that complication for this example.

Predicted Values (cont.)

If $\hat{\mu}$ is the result of `predict.aster.formula` without the optional argument `amat`, then when the optional argument `amat` is given it does parameter estimates with standard errors for a new parameter

$$\hat{\zeta} = A^T \hat{\mu},$$

where A is a known matrix (the `amat` argument).

Since we want 7 confidence intervals A^T has 7 rows, and since μ is length 63, A^T has 63 columns. Thus A is a 63×7 matrix.

Fairly simple, except now comes some serious bad design.

Predicted Values (cont.)

Quoted from `help(predict.aster)`

> For `predict.aster`, a three-dimensional array with `dim(amat)[1:2] == dim(modmat)[1:2]`.
>
> For `predict.aster.formula`, a three-dimensional array of the same dimensions as required for `predict.aster` (even though `modmat` is not provided). First dimension is number of individuals in `newdata`, if provided, otherwise number of individuals in `object$data`. Second dimension is number of variables (`length(object$pred)`).

Also clear as mud.

Predicted Values (cont.)

So here is another description. The argument `amat` is a three dimensional array.

The first dimension is the number of individuals (in `newdata` if provided, and otherwise in the original data).

The second dimension is the number of nodes in the graph.

The third dimension is the number of parameters we want point estimates and standard errors for.

Predicted Values (cont.)

Let a_{ijk} denote the elements of this array, and let μ_{ij} denote the elements of the result of calling `predict.aster.formula` without the `amat` argument, these elements being stuffed into a matrix columnwise as we showed back on slide 99. Then we are trying to estimate the parameter vector having components

$$\zeta_k = \sum_{i=1}^{n_{\text{ind}}} \sum_{j=1}^{n_{\text{node}}} a_{ijk} \mu_{ij}$$

Predicted Values (cont.)

```
> npop <- nrow(fred)
> nnode <- length(vars)
> amat <- array(0, c(npop, nnode, npop))
> dim(amat)

[1] 7 9 7
```

We want only the means for the k-th individual to contribute to ζ_k. And we want to add only the headcount entries.

```
> foo <- grepl("hdct", vars)
> for (k in 1:npop)
+     amat[k, foo, k] <- 1
```

This three-way array is too big to print on a slide. We'll just try it out.

Predicted Values (cont.)

```
> pout.amat <- predict(aout, newdata = renewdata,
+      varvar = varb, idvar = id, root = root,
+      se.fit = TRUE, amat = amat)
> pout.amat$fit

[1] 2.375984 1.502068 1.565962 1.064478 1.800250
[6] 2.499468 1.705529

> rowSums(herman)

      AA   Eriley        Lf  Nessman     NWLF      SPP
2.375984 1.502068 1.565962 1.064478 1.800250 2.499468
 Stevens
1.705529
```

Hooray! They're the same!

Predicted Values (cont.)

There is an important lesson here.

When doing something complicated, do it first in the most obvious most transparent way that is most likely to be correct.

Then when we do it a trickier way (as we had to do to get standard errors), we can be sure it too is correct when it gets the same results as the transparently correct way.

If you don't understand what you are doing, then it is probably wrong.

Predicted Values (cont.)

```
> foo <- cbind(pout.amat$fit, pout.amat$se.fit)
> rownames(foo) <- as.character(fred$pop)
> colnames(foo) <- c("estimates", "std. err.")
> round(foo, 3)
```

	estimates	std. err.
AA	2.376	0.446
Eriley	1.502	0.196
Lf	1.566	0.249
Nessman	1.064	0.309
NWLF	1.800	0.182
SPP	2.499	0.289
Stevens	1.706	0.222