I. What are MARK and RMark?

a. *What is MARK and what does it do?*

MARK is freely available software that was written by Dr. Gary White at Colorado State University (http://welcome.warnercnr.colostate.edu/~gwhite/mark/mark.htm, http://www.phidot.org/software/mark/index.html). MARK was originally designed to provide a common graphical user interface (GUI) and FORTRAN code to replace the myriad of software programs (JOLLY, CAPTURE, SURVIV, BROWNIE, ESTIMATE) for analysis of various types of capture-recapture data to provide estimates of survival and abundance. MARK has since expanded to include occupancy (absence/presence) models, nest survival models and some other types of analysis that are not strictly capture-recapture but are similar in structure. Dr. White continues to add new models and new features to MARK.

MARK is comprised of two primary programs: Mark_Int.exe and mark.exe. The user interacts with the first program which is the GUI that is used to design models and review and manipulate the results (e.g., goodness of fit, plots, residuals, model averaging, etc). The second is the FORTRAN program which is run behind the scenes to fit models to data to obtain estimates of the model parameters and their precision and related derived values (e.g., abundance, lambda). When a model is "run" from the interface, a text file (*.inp) is created which mark.exe reads and uses to construct the model and estimate the parameters, etc. It creates 3 output files (*.out, *.vcv, *.res) which are read in by the GUI and stored in the .dbf and .fpt project files that MARK constructs for each project. Once the output files are stored, they are deleted.

In a few cases, model building in MARK can be accomplished by simply selecting from a set of pre-defined models. But in many real examples, a model is built by constructing parameter information matrices (PIMS) and a design matrix (DM). Some very nice tools to construct PIMs and DMs are provided in the GUI but ultimately they must be constructed manually by the user with the exception of the pre-defined models. This manual model construction can become very tedious and error-prone with large datasets and complex analysis. Also, if the data structure changes (e.g., group structure or additional capture occasion) a new MARK project file must be created and all of the models (PIMs and DMs) must be reconstructed manually. This is particularly problematic for any monitoring project which collects new data each year and requires recreating the models manually each year to update the models. While MARK is very useful and capable software, it can consume large amounts of time and require more patience than I have; thus, the motivation for RMark.

b. What is RMark and what does it do?

RMark is a software package for the R computing environment that was designed as an alternative interface that can be used in place of MARK's GUI (Mark_Int.exe) to describe models with formula so they need not be constructed manually. RMark uses the R language and tools to construct user-specified models that are written into input files (*.inp) and then runs mark.exe to fit the model to the data and estimate parameters, precision, etc. RMark then extracts results from the output files (*.out and *.vcv) but does not delete those files and instead stores the output filenames in R objects so they can be referenced later. RMark does some computation for calculation of real

parameters, covariate predictions, model averaging and variance components. BUT, RMark DOES NOT fit models to data! It uses MARK to do that. RMark builds models that MARK fits to the data.

c. *Advantages/Disadvantages of using RMark*

Development of software user interfaces over the last 3 decades has largely been focused on graphical user interfaces (GUI). GUIs can be relatively easy to use and learn and they are great for applications such as word processing. However, they are not always ideal for scientific applications which should be easily repeatable and well-documented. Unless the GUI creates a product such as a script/macro or other result that can be documented and easily reproduced tracking down errors or replicating an analysis can be difficult or impossible. Command interfaces like the R computing environment are more demanding for the user and often more difficult to learn but they provide the scripting tools necessary to be able to document and automate analysis which can then be easily replicated. After learning R you will come to appreciate, its flexibility and power. If you are diligent about using scripts and adding documentation, you can feel comfortable about picking up the script at some point in the future and knowing exactly what you did and you could easily re-run the analysis if you misplaced results or wanted to change a plot or found a mistake. RMark was written for R to take advantage of scripting and the tools for automating model development, but also to set it in an environment that has a rich set of other tools for data manipulation, computation and graphics. To some this will be a disadvantage of using RMark due to the necessity of learning R but hopefully after the R portion of the workshop you will feel more comfortable with R and begin to understand how useful R can be for many different analysis tasks beyond capture-recapture analysis.

Beyond scripting, the primary advantage of RMark is the automation of model development. A simple analysis can be done with 2 function calls to import the data and run a model. More complex analysis may involve creating commands for data manipulation before running a set of models. However, with RMark there is no manual construction of PIMs or DMs and no manual labeling of parameters and output. That is all done automatically which means that the user can focus on the analysis without spending countless hours at the keyboard building PIMS and DMs. In addition to saving time, the model automation means the analysis is less prone to errors from an errant keyboard entry or by misaligned PIMS from incorrectly dragging PIMS on the PIM chart. This does NOT mean that model development in RMark is error free. You can make mistakes if you are not careful. Also it doesn't mean that you can develop models for MARK without understanding how they are constructed and what they do. But once you learn RMark, you will be able to analyze and document your analysis of capture-recapture data in far less time than with the MARK GUI. If you expect that you will only analyze one simple capture-recapture data set in the near future, use MARK and forget about RMark unless you want to learn R for other reasons. However, if you are responsible for analyzing capture-recapture data in a monitoring role or you analyze many different capture-recapture data sets, then learning RMark will be worthwhile.

At present, the RMark interface does not replicate every aspect of the MARK interface. In particular, not every model in MARK is supported by RMark. Also, at present RMark does

not use the residuals file or provide any way to examine them nor does it provide code computation of the median chat for over-dispersion. To circumvent this problem, import and export functionality is included in RMark which provides the opportunity to export models to the MARK GUI interface to use any capabilities that it has that are not included in RMark.

d. *Supported models*

At the time of writing, RMark supports the following MARK models: Cormack-Jolly-Seber (CJS), recovery, Burnham-Barker live/dead, Burnham Jolly-Seber, Pradel Jolly-Seber, closed capture, POPAN, known-fate, multistrata, robust design, nest survival and occupancy. Some of these models have slightly different aspects to them because of the type of data (e.g., nest survival, known fate and occupancy) or due to variation in the model structure (e.g., multistrata and robust design). However, in general, there is little variation in your use of RMark to analyze data with these various models. After learning to use RMark with one of these types of models, you should be able to transition to other models easily. Because I have a limited amount of time for the workshop, I am going to focus on CJS models and if I get time I'll discuss and give examples of other models as well.

II. MARK/RMark help, documentation, and example datasets

a. *Cooch and White online book for MARK; Appendix C for RMark*

A limited amount of material can be covered in the time allotted for the workshop. However, once you finish with the workshop and you are back in your office analyzing some data, there are a number of available resources for you to use to answer questions and to learn more about capture-recapture analysis, MARK and RMark. If you haven't already done so, download a copy of the electronic book by Evan Cooch (at Cornell) and Gary White (http://www.phidot.org/software/mark/docs/book/). You can download the entire book by clicking on the "one single file" link. Within the 800 pages of material it contains a wealth of information on capture-recapture analysis and the use of MARK. The book shows numerous examples with various datasets. To download those datasets, click on the down-arrow where it says "select chapter" under "Book chapters and data files" on the left side of the web page. The selection for data files is at the very bottom. You will not need these data files for RMark because all of the example data files for RMark are included with the package; however, you can convert the data files using an RMark function (`convert.inp`) and use them with RMark to see if you can get the same results from RMark as they show with MARK.

Appendix C of the electronic book contains 100+ pages dedicated to RMark and a very brief R primer at the end of the appendix (C.24). Even if you have a reasonable grasp of R after the workshop, it may be useful to review the tutorial to understand how lists provide useful structures for working with models in RMark. Appendix C provides much of the material we will discuss in the workshop and it provides some examples for various models that we may not be able to cover in the workshop. The documentation for RMark was intentionally included in the book as an appendix to make it clear that you first need a solid grounding in capture-recapture analysis with MARK to fully understand and use RMark. Most of the errors you will make will be R syntax errors (e.g., forgetting a comma

or using ")" instead of "]", etc.). However, some of the errors will be from the RMark code and some of the more common errors are described in C.23 of the appendix.

b. *R Help Files for RMark; What's New?*
Appendix C is written as a user guide rather than as a reference book and it does not cover every function in RMark. Nor does it describe every argument of the functions that are covered. For a reference "book" that documents each function and the example datasets used in RMark, you will need to access the help files written for RMark. They can be accessed in R with a "?" followed by a function name (e.g., `?mark`) or with `help.search("some text")`. The complete set of help files can also be accessed via the file RMark.chm which you will find in the RMark/chtml sub-directory that is contained under the Library sub-directory for R which is typically stored under the Program Files directory. Also, for this workshop a pdf version (RMarkHelpFiles.pdf) has been placed on the workshop web pages (myUSGS).

c. *Phidot list server*
Another resource for help is the phidot forum (http://www.phidot.org/forum/index.php) which is a user support group that contains sections for MARK, SURGE, PRESENCE and other software packages. I recommend joining the forum and I highly recommend selecting the option to have messages emailed to you. It generates very little email but I use it to announce important revisions to RMark. If you have a question about RMark, you can send it to me (jeff.laake@noaa.gov) but I prefer that the messages be sent through the phidot forum because other users may be able to answer it (reducing the load on me). Even if I do answer it, the forum provides an archive for others to search to find an answer to a question that may have already been asked. So, first search the archive and if you can't find an answer, post your question or problem (to the Program MARK - statistics and analysis help section). If you have a question/problem, then others may also, so please don't be afraid to post. If I think the answer may take some back-and-forth, I may answer you directly off-list for an email exchange and then we can post a summary of our exchange on the forum.

III. Elements of RMark capture-recapture data

a. *Description of Dipper data; data(dipper)*
Throughout the workshop we will use a well-known set of data collected on European dippers (Lebreton et al 1992; Ecological Monographs). The dipper data are included in the RMark package and as with any example dataframe in R they can be retrieved into the workspace by typing: `data(dipper).` A description of an example dataframe in R can be obtained by typing a ? followed by the name of the data (e.g., `?dipper`). In the help file for dipper, you will see a description of the dataframe and some details and example code describing its use with RMark. In the help you will see that dipper contains 294 records and it has 2 columns (fields) named `ch` and `sex`. If you type `summary(dipper)`, you should see the following:

```
> summary(dipper)
      ch                 sex
 Length:294         Female:153
 Class :character   Male  :141
 Mode  :character
```

b. *Capture (encounter) history*

The field `ch` is the capture (encounter) history which is the essential observational data for capture-recapture. In all RMark dataframes, the capture history must be named `ch` and it must be character data. You can assess that it contains character data if you look at `summary` as above or use `str` (structure function) which also shows the first 5 data values:

```
> str(dipper$ch)
 chr [1:294] "0000001" "0000001" "0000001" "0000001" "0000001" ...
```

The typical capture history is a string of "0" or "1" values where "0" means the animal was not captured (or encountered) and a "1" means that the animal was captured. The positions in the string represent sampling occasions which are ordered from left to right in their temporal order (i.e., occasion 1, occasion 2, …, occasion k). Not all capture histories are composed of 0s and 1s and the structure of the capture history depends on the capture-recapture model. For example, with a Multistrata model a capture history can be composed of letters (e.g., A0BBA00B) and for a live/dead model each sampling occasion is represented by 2 positions in the capture history (e.g., 100001 represents 3 occasion and intervals).

Even when the capture history has identical structure, proper interpretation of a capture history can depend on the capture-recapture model in more subtle ways. For example, we will focus in this workshop on the Cormack-Jolly-Seber (CJS) type of models. For CJS models, the first "1" in the capture history represents a release of an animal and not strictly a capture. Technically for a CJS model it should be referred to as a release-recapture history because it is only the data for occasions after the first "1" (the release) that are included in the model. Thus, it is only recaptures that are modeled which is why it is labeled "Recaptures only" in the MARK interface. In contrast, if you were to analyze the same capture history with a Jolly-Seber model, the value for each occasion (each position in the string) is used in the model and not just the values after the first "1".

c. *Frequency (optional)*

In the dipper data, each record (row) in the dataframe represents a single animal. It is possible to aggregate animals that have the same capture history into a single record (row) and to assign a value that represents the number of animals with that capture history. To do so, the dataframe should have a column named `freq` which is a numeric field which contains the frequency (count) of animals with that particular capture history. If there is no column named `freq` then it is assumed to be 1 for each entry.

d. *Groups (factor variables)*

The only required column in a dataframe for RMark is `ch` and any other columns are optional. However, for most analysis there will be ancillary data (variables) about the animals and we will often want to know how those variables might affect (co-vary with) the parameters (e.g., survival) that we are estimating. These variables (covariates) are attributes about the **animals** and thus they are called "individual covariates" with

"individual" meaning that they can vary from animal to animal. These covariates are either qualitative or quantitative and can be either static or dynamic with respect to time. Qualitative (nominal) variables are often referred to as categories or classes and in R they are called factor variables. In RMark, factor variables are used to define "groups" of animals for MARK. For example, in the `dipper` data, the field `sex` is a factor variable with character values "Female" and "Male". This factor variable will be used to define 2 groups in the data for RMark which allows specification of parameters to differ by group. The actual value of `sex` is numeric with values 1 and 2 and the printed character values for factor variables are called levels. The clarification between levels and values can be seen with the `str` and `levels` functions:

```
> str(dipper$sex)
 Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
> levels(dipper$sex)
[1] "Female" "Male"
```

The levels of a factor can be modified with the `levels` function. For example, we can rename the levels "F" and "M" as follows:

```
> levels(dipper$sex)=c("F","M")
> str(dipper$sex)
 Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 1 ...
```

The ordering of factor levels is alphanumeric ("A" before "B" and "1" before "2" and "1" before "A"). However, you can change the order of levels with the `factor` function or the `relevel` function:

```
> dipper$sex=factor(dipper$sex,levels=c("M","F"))
> str(dipper$sex)
 Factor w/ 2 levels "M","F": 2 2 2 2 2 2 2 2 2 2 ...

> dipper$sex=relevel(dipper$sex,"F")
> str(dipper$sex)
 Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 1 ...
```

The `relevel` function simply redefines which level is first in the order; whereas, the `factor` function can be used to completely change the order. Note above that in each case the data listed for the first few records of dipper are always females but when "F" is first they are represented by 1 and when "M" was first they had the value 2 because "F" was the second in the list. It is useful to understand these commands because the levels are used as labels in the MARK and RMark output and the first level is used as the intercept in the model. Sometimes it is helpful to be able to change these values.

Factor variables used to define groups are static variables which are constant across time (no dipper sex changes!). Factor variables can also be analyzed as a set of numeric dummy variables but this will be covered later.

e. Individual covariates (numeric variables)

Even though factor variables are covariates for individual animals, for RMark they are called group or factor variables and the name "individual covariates" is reserved for numeric variables (e.g., weight) due to the manner in which they are treated in MARK. An individual covariate is any numeric (quantitative) variable that is measured for each animal in the data. Most individual covariates are static with one measurement (e.g., initial weight at time of release) that does not vary over time. It is possible to specify time-varying individual covariates by specifying separate covariates for each time (e.g., var1, var2, …, vark for some variable named var for each of the k occasions). Time-varying individual covariates are rarely used because it is rare to know the covariate value over time for each animal regardless of whether it was caught or not. There are some exceptions including age because once you know an animal's initial age, you know its value for each time afterward. However, it typically isn't necessary to treat age as a time-varying covariate as we will explain later. We'll see more on individual covariates later.

f. Another example; data(example.data)

Before we leave the topic of data and data structure, let's look at one more example included with RMark which has the clever name `example.data`. It is a simulated set of data that was created solely to demonstrate some of the features of RMark. It also has a help file that contains some example code for RMark. Here we only want to demonstrate a data set with both factor and individual (numeric) covariates:

```
> data(example.data)
> summary(example.data)
      ch                weight         age      sex      region
 Length:6000        Min.   : 0.1820   1:2000   F:3000   1:1500
 Class :character   1st Qu.: 7.9894   2:2000   M:3000   2:1500
 Mode  :character   Median : 9.9666   3:2000            3:1500
                    Mean   : 9.9751                     4:1500
                    3rd Qu.:11.9996
                    Max.   :21.1301
```

Notice that the format is similar for the 3 factor variables: `age, sex` and `region`. For factor variables, the `summary` function shows the number of records for each level of the factor in order. In contrast, the numeric variable `weight` is summarized as the min, max, mean and quartiles. This is an easy way to distinguish numeric from factor variables. This data is also useful to illustrate that more than one factor variable can be used to define groups. See `?example.data`. There are many different example data sets that have been included with the RMark package for the various types of capture-recapture models.

IV. A simple overview example

a. Cormack-Jolly-Seber(CJS) model background

We will be primarily discussing CJS models in the workshop. To get everyone up to speed, I'll give a very brief description of the structure of CJS models using the dipper data. Then I'll briefly show some analyses of the dipper data with the MARK GUI interface and then jump into RMark.

First of all, it is important to recognize that CJS models are for open populations in which animals are allowed to die (and possibly emigrate). With CJS models we are typically interested in estimating survival which is represented by the Greek letter Phi (ϕ). Think about the process of releasing animals with some unique way to identify them and then recapturing (or resighting) some of them afterwards at some set of occasions. It is obvious that if you do recapture the animal at an occasion that it survived to that occasion. Also, if you don't see an animal on a particular occasion but you see it on a later occasion then you know it was alive and you simply failed to capture it. For example, if the capture history with 7 occasions is 1001100 it means that we released it on occasion 1 and it was recaptured on occasions 4 and 5 but was not captured on occasions 2, 3, 6 and 7. We know it lived until occasion 5 and it was simply missed on occasions 2 and 3. But we don't know what happened on occasions 6 and 7 but we do know that one of the following three things happened:

1) It lived until occasion 7 and we missed it on occasion 6 and 7, or
2) It died in the interval between occasions 6 and 7 and we missed it on occasion 6, or
3) It died sometime after it was last seen between occasions 5 and 6.

It is these strings of 0s at the end of the capture history where we have incomplete knowledge about the fate of the animal, that illustrate the need to estimate the probability of (re)capturing animals (p) to be able to estimate survival (ϕ) which is the parameter of interest.

Consider capture histories for animals released on the first occasion in the dipper data which has 7 sampling occasions. The first occasion is simply a release of the first cohort. Thus, we have 6 possible recapture occasions for this cohort of animals and there are 6 intervals between the occasions. If we thought that the probability of capturing animals (p) was different for each occasion and the probability of surviving the interval between occasions was different for each interval, then we would need 6 ps and 6 ϕs and we could number these p_2, p_3, p_4, p_5, p_6, p_7 and ϕ_1, ϕ_2, ϕ_3, ϕ_4, ϕ_5, ϕ_6 to label them. If x_2, x_3, x_4, x_5, x_6, x_7 are the 0/1 outcomes at each of the recapture occasions 2-7 we can schematically represent the relationship between the data and parameters as follows:

$$\begin{array}{cccccccccccccc} & & p_2 & & p_3 & & p_4 & & p_5 & & p_6 & & p_7 \\ 1 & ----- & \phi_1 & ------ & x_2 & ----- & \phi_2 & ------ & x_3 & ----- & \phi_3 & ------ & x_4 & ----- & \phi_4 & ------ & x_5 & ----- & \phi_5 & ------ & x_6 & ----- & \phi_6 & ------ & x_7 \end{array}$$

The probability that an animal was recaptured on each of the subsequent 6 occasions after release (ch=1111111) is the product of each of the parameters: $\phi_1 p_2 \phi_2 p_3 \phi_3 p_4 \phi_4 p_5 \phi_5 p_6 \phi_6 p_7$. If an animal was not captured on occasion 2 and 6 (ch=1011101), the probability would be $\phi_1(1-p_2)\phi_2 p_3 \phi_3 p_4 \phi_4 p_5 \phi_5(1-p_6)\phi_6 p_7$. Now let's consider the capture history we mentioned earlier ch=1001100. The probability for the first part of the capture history until the last 1 is $\phi_1(1-p_2)\phi_2(1-p_3)\phi_3 p_4 \phi_4 p_5$. For the last part of the capture history (00) there are 3 possible explanations as outlined above and the probabilities are:

1) $\phi_5(1-p_6)\phi_6(1-p_7)$
2) $\phi_5(1-p_6)(1-\phi_6)$

3) $(1-\phi_5)$

So the probability for the entire capture history is:

$\phi_1(1-p_2)\phi_2(1-p_3)\phi_3p_4\phi_4p_5\ [\phi_5(1-p_6)\phi_6(1-p_7) + \phi_5(1-p_6)(1-\phi_6) + (1-\phi_5)]$

The probabilities for any of the other possible observed capture histories are constructed in a similar fashion. It is useful to understand how these probabilities are put together but you will never have to do that yourself because MARK does that for you in the process of fitting the model to the data. What you do need to understand is how the parameters relate to your data and how to specify specific models for the parameters. But before we get there, let's extend the parameter notations for the complete set of dipper data.

For the remainder of the release cohorts, the schematic representation would be as follows:

			p_3		p_4		p_5		p_6		p_7	
0	----------	1	----- ϕ_2 ------	x_3	----- ϕ_3 ------	x_4	----- ϕ_4 ------	x_5	----- ϕ_5 ------	x_6	----- ϕ_6 ----	x_7
					p_4		p_5		p_6		p_7	
0	----------	0	----------	1	----- ϕ_3 ------	x_4	----- ϕ_4 -------	x_5	----- ϕ_5 ------	x_6	----- ϕ_6 ----	x_7
							p_5		p_6		p_7	
0	----------	0	----------	0	----------	1	----- ϕ_4 -------	x_5	----- ϕ_5 ------	x_6	----- ϕ_6 -----	x_7
									p_6		p_7	
0	----------	0	----------	0	----------	0	----------	1	----- ϕ_5 --------	x_6	----- ϕ_6 ----	x_7
											p_7	
0	----------	0	----------	0	----------	0	----------	0	----------	1	------- ϕ_6 ----	x_7

If there was a seventh release cohort, it would not add information to the existing CJS model because there would be no recaptures possible.

Now if we were to just write down the indices for these sets of parameters in a matrix form they would look as follows:

Phi (ϕ)

1	2	3	4	5	6
	2	3	4	5	6
		3	4	5	6
			4	5	6
				5	6
					6

p

2	3	4	5	6	7
	3	4	5	6	7
		4	5	6	7
			5	6	7
				6	7
					7

If we wanted to refer to each parameter by a number such that we could distinguish Phi from p we might represent them as follows:

Phi (ϕ)

1	2	3	4	5	6
	2	3	4	5	6
		3	4	5	6
			4	5	6
				5	6
					6

p

7	8	9	10	11	12
	8	9	10	11	12
		9	10	11	12
			10	11	12
				11	12
					12

These matrices are PIMs – Parameter Index Matrices – that are used in MARK. These are called time PIMS because the indices vary across time for both Phi and p. Note that p is represented by the recapture occasion number whereas Phi is represented by the first occasion of the interval (e.g., 2 for survival for the interval between occasion 2 and 3). You can think about the number being an "index" to a list of parameters as shown below.

Index	Parameter	Occasion/Interval
1	Phi	1
2	Phi	2
3	Phi	3
4	Phi	4
5	Phi	5
6	Phi	6
7	p	2
8	p	3
9	p	4
10	p	5
11	p	6
12	p	7

So far we have only considered that parameters might vary by time but what if we also thought that parameters might vary for each release cohort? Then we would have to have separate parameters for each row as well as each column and we might use what

are called all-different PIMs in MARK because each number (index) in the matrix is different:

Phi (ϕ)

1	2	3	4	5	6
	7	8	9	10	11
		12	13	14	15
			16	17	18
				19	20
					21

p

22	23	24	25	26	27
	28	29	30	31	32
		33	34	35	36
			37	38	39
				40	41
					42

Each row as a cohort from 1 to 6 and each column represents an occasion and each parameter number is an index into a list which is now expanded:

Phi			p		
Index	Cohort(row)	Interval	Index	Cohort (row)	Occasion
1	1	1	22	1	2
2	1	2	23	1	3
3	1	3	24	1	4
4	1	4	25	1	5
5	1	5	26	1	6
6	1	6	27	1	7
7	2	2	28	2	3
8	2	3	29	2	4
9	2	4	30	2	5
10	2	5	31	2	6
11	2	6	32	2	7
12	3	3	33	3	4
13	3	4	34	3	5
14	3	5	35	3	6
15	3	6	36	3	7
16	4	4	37	4	5
17	4	5	38	4	6
18	4	6	39	4	7
19	5	5	40	5	6
20	5	6	41	5	7
21	6	6	42	6	7

We will refer to these lists as “design data” (or design data list; ddl) and they will be expanded further to include age, group factor variables, and any other variables that are relevant to a particular study and model.

b. An analysis of the dipper data in MARK

This portion of the workshop will be done interactively with MARK. Because RMark has to satisfy all the requirements of MARK, it is useful to see a demonstration of MARK even if

you never use it again. Also, if you have not used MARK before, it should give you an appreciation for how much time you will save with RMark. You may wonder how MARK derives the estimates of the parameters from the data (capture histories) and the specification of the parameters. The answer depends in part on the type of model (e.g., CJS, JS, closed capture etc); however, in general the parameter estimates for most of the model types are derived via maximum likelihood and the likelihood of the data is based on the probabilities for particular capture histories as a function of the parameters. For example, for a CJS model with *n* capture histories where π_i is the probability of the ith capture history, the maximum likelihood estimates of the vector of parameters (**p** and **ϕ**) are the values that maximize the following log-likelihood function:

$$\mathcal{L}(\boldsymbol{\Phi}, \boldsymbol{p}) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{log}(\pi_i)$$

The number of p and ϕ parameters depends on structure of the model (e.g., constant, time-dependent, age-dependent, etc) and the value of π_i in terms of the parameters depends on the observed capture history as shown earlier. To learn more about maximum likelihood estimation for capture-recapture models, see White et al. (1982) (http://welcome.warnercnr.colostate.edu/class_info/fw663/).

We will delve into more detail later but for now I just want to introduce the terms link function and beta and real parameters so you know what they are. Many of the parameters in capture-recapture are probabilities that are bounded between 0 and 1 and other parameters are strictly positive values like abundance. When MARK is fitting models to data it is finding the maximum of the likelihood function with respect to a set of parameters. Maximization is one form of optimization and optimization problems are generally easier if there are no bounds on the range for the optimization (e.g., parameters). To avoid bounded optimization with the real parameters (e.g., Phi, p, etc), a link function is used which transforms the real parameters to a set of unbounded beta parameters. The likelihood is maximized with respect to the beta parameters and because there is a one-to-one transformation between the beta and real parameters, the maximum for the beta parameters corresponds to the maximum for the real parameters. For example, the logit link function is often used for probabilities and an example for capture probability might look as follows:

$$p = \frac{e^{\beta}}{1 + e^{\beta}} = \frac{1}{1 + e^{-\beta}}$$

$$\beta = \log\left(\frac{p}{1 - p}\right)$$

The top equation is the inverse link function which computes the real parameter from the beta parameter. The second equation is the link function which computes the beta parameter from the real parameter. Both forms of the top equation will work but the far right equation should be used because it is more numerically stable. Note that the log function used here is the natural logarithm (base e) and not base 10. If you use `log(x)` in

R it will be the natural logarithm and `log10(x)` is base 10. However, if you happen to use Excel `log(x)` is base 10 and `ln(x)` is base e. As an example, if p=0.5 then

$$\beta = \log\left(\frac{0.5}{1-0.5}\right) = \log(1) = 0$$

and

$$p = \frac{e^0}{1+e^0} = \frac{1}{1+e^{-0}} = \frac{1}{2} = 0.5$$

As β becomes a large positive number p approaches 1 and as β becomes a large negative number p approaches 0. Note that large is relative here and even a number like +5 or -5 is close to the boundary with p= 0.993 and 0.007 respectively. In R, the `plogis(x)` function provides an easy way to compute the inverse logit function (e.g., `plogis(-5)`).

c. How to "start" RMark

RMark is an R package which extends the software available in R. I'm presuming that you will have installed R, MARK and RMark as explained on the Seattle09 website. Also, by now you should have created a sub-directory for the workshop that contains all of the files you'll need. Navigate to the location where you stored the workshop files. Start up R by double-clicking on the .Rdata file in that directory. If you don't have a .Rdata file yet, start R and use File/Save Workspace, navigate to your directory and click Save. Type `library(RMark)` and you should see something like the following on your screen:

```
> library(RMark)
This is RMark 1.8.9
Built: R 2.8.1; i386-pc-mingw32; 2009-03-09 17:06:29; windows
```

For each new R session you need to type `library(RMark)` if you want to use it during the session. Alternatively, you can store that command in your RProfile.site file so the package will be attached automatically for each session. If there are any folks with a 64-bit Vista operating system then you'll have to also add `MarkPath="C:/Program Files (x86)/Mark/"` to your RProfile.site file. You would also need to do something similar if you chose to store MARK in a directory other than the default.

d. Running a simple model

Let's start with RMark by using the `mark` function to fit to the dipper data a very simple model with constant survival and constant capture probability and storing it into an object we will call `myfirstmodel`. You should see the following on your monitor:

```
> data(dipper)
> myfirstmodel=mark(dipper)

Output summary for CJS model
Name : Phi(~1)p(~1)

Npar :  2
-2lnL:  666.8377
AICc :  670.866

Beta
                 estimate        se       lcl       ucl
Phi:(Intercept) 0.2421484 0.1020127 0.0422035 0.4420933
p:(Intercept)   2.2262658 0.3251093 1.5890516 2.8634801
```

```
Real Parameter Phi

         1        2        3        4        5        6
1 0.560243 0.560243 0.560243 0.560243 0.560243 0.560243
2          0.560243 0.560243 0.560243 0.560243 0.560243
3                   0.560243 0.560243 0.560243 0.560243
4                            0.560243 0.560243 0.560243
5                                     0.560243 0.560243
6                                              0.560243

Real Parameter p

          2         3         4         5         6         7
1 0.9025835 0.9025835 0.9025835 0.9025835 0.9025835 0.9025835
2           0.9025835 0.9025835 0.9025835 0.9025835 0.9025835
3                     0.9025835 0.9025835 0.9025835 0.9025835
4                               0.9025835 0.9025835 0.9025835
5                                         0.9025835 0.9025835
6                                                   0.9025835
```

So what did the `mark` function do with its argument dipper? It assumed by default that this was a "CJS" model and based on that it "processed" the data, which I'll explain in more detail later. Then it set up some default "design data", and based on the default model of constant Phi and p, it created the design matrix and the complete input file for MARK. Then it ran MARK with the input file and extracted the results from the output files. Finally, it produced a summary of the results that you see on your screen and it compiled all the results into a list which it retuned as its value and that was subsequently assigned (stored) to my `firstmodel`. The assignment operator = is the same as <- in all but some special circumstances. I use = exclusively but note that == is a test for equality and not assignment.

The summary provides a descriptor of the model, number of parameters, negative 2 * the log likelihood value, AICc, and a listing of the beta and real parameters. For this model the real Phi and p can be computed using the `plogis` function:

Phi

```
> plogis(0.2421484 )
[1] 0.560243
```

p

```
> plogis(2.2262658)
[1] 0.9025835
```

Note that even though there are 21 Phi and 21 p real parameters shown, there is only one unique Phi and one unique p. The number of parameters in the model is the number of betas and each beta will correspond to a column in the design matrix. MARK finds the maximum likelihood estimates for the betas (i.e., optimizes the likelihood with respect to values of betas) and the real parameter estimates are computed with the beta estimates.

e. Accessing and locating MARK output files

If this is the first time you used the `mark` function in this directory it would have created a file named mark001.inp which contains the data and model structure input for MARK. As it creates more models the numeric counter is increased (i.e., `mark002`, `mark003`). When `mark.exe` was run with the `mark001.inp` file, it created 3 more files:

mark001.out, mark001.vcv and mark001.res. You can see a listing of the files in your current directory by using the function `list.files()` and depending on what other files you have included in the directory it may look something like this:

```
> list.files()
[1]"ClassScript.R"      "dipper.txt"         "mark001.inp"
   "mark001.out"        "mark001.res"        "mark001.vcv"
```

I show you these files just to let you know they are there and how they are named but you never have to work with them directly or remember how they are numbered or which is which. However, you don't want to delete them either if you want to continue using the results from the models. Later I'll show you how to delete unused files. For now, I'll simply show you that if you type the name of the model object `myfirstmodel` it will open up a notepad window containing the output file. RMark knows which file to show because `myfirstmodel$output` contains the prefix portion of the filename:

```
> myfirstmodel$output
[1] "mark001"
```

Even though you can look at the MARK output file from RMark, you will rarely want to do so because all of the results are stored in the R object (e.g., `myfirstmodel`) and RMark provides various ways to view and use those results in R. In addition, viewing the results in the MARK output file can be misleading because the labeling in the output file is not always clear but this is remedied in the RMark output. I'll explain more on that later.

f. Structure of a mark object in R

RMark provides functions to extract portions of the output stored in the R object (e.g., `myfirstmodel`) but not all of the results. Thus, it is useful to understand the structure of the resulting object. Each object created by `mark` is a list and it has two class values with the first being `"mark"` and the second being the type of mark model:

```
> mode(myfirstmodel)
[1] "list"
> class(myfirstmodel)
[1] "mark" "CJS"
```

Sometimes I will use the term "mark" object to refer to an R object with class="mark". To look at the structure of any R list object, simply use `names(object)`:

```
> names(myfirstmodel)
 [1] "data"             "model"            "title"            "model.name"
 [5] "links"            "mixtures"         "call"             "parameters"
 [9] "time.intervals"   "number.of.groups" "group.labels"     "nocc"
[13] "begin.time"       "covariates"       "fixed"            "design.matrix"
[17] "pims"             "design.data"      "strata.labels"    "mlogit.list"
[21] "profile.int"      "simplify"         "model.parameters" "results"
[25] "output"
```

Most of the elements of the list are single elements or vectors of values that are used to describe the model and its structure or options like:

```
> myfirstmodel$model
[1] "CJS"
```

```
> myfirstmodel$nocc
[1] 7
```

which are the type of capture-recapture model and the number of occasions which is 7 for the dipper data. You'll be most interested in "`results`" which is also a list of the results that were extracted from the output. You can examine these by simply typing:

```
> myfirstmodel$results
$lnl
[1] 666.8377
$deviance
[1] 58.15788
$npar
[1] 2
$n
[1] 426
$AICc
[1] 670.866
$beta
                estimate        se       lcl       ucl
Phi:(Intercept) 0.2421484 0.1020127 0.0422035 0.4420933
p:(Intercept)   2.2262658 0.3251093 1.5890516 2.8634801
$real
                 estimate        se       lcl       ucl fixed    note
Phi g1 c1 a0 t1 0.5602430 0.0251330 0.5105493 0.6087577
p g1 c1 a1 t2   0.9025835 0.0285857 0.8304826 0.9460113
$beta.vcv
             [,1]         [,2]
[1,]  0.010406594 -0.008507172
[2,] -0.008507172  0.105696072
$derived
data frame with 0 columns and 0 rows
$derived.vcv
NULL
$covariate.values
NULL
$singular
NULL
$real.vcv
NULL
```

Not every entry is used for each model. For example, `singular` lists the indices of parameters that are at boundaries or confounded as determined by MARK. If there are no such parameters then its value will be NULL. You can test for a NULL value using the `is.null` function:

```
> is.null(myfirstmodel$results$singular)
[1] TRUE
```

Other values such as `real.vcv` are only completed if an optional argument is set to TRUE.

g. Summary results from a mark object

The results shown after creating `myfirstmodel` with the dipper data were created by the function `summary.mark` which was run by default from `mark`. You can see the same results again by typing `summary(myfirstmodel)`. Generic functions like

summary, print, coef look at the class of the object and in this case call the functions summary.mark, print.mark and coef.mark when they are used with an object with class="mark". The function summary.mark has various arguments that can be used to control what and how the information is summarized. We'll see more of this in the first exercise.

h. *Extracting coefficients and other elements of a mark object*
If you only want to look at the estimates of the beta parameters, this is most easily done with the coef function:

```
> coef(myfirstmodel)
                 estimate        se       lcl       ucl
Phi:(Intercept) 0.2421484 0.1020127 0.0422035 0.4420933
p:(Intercept)   2.2262658 0.3251093 1.5890516 2.8634801
```

The coef function returns a dataframe containing the estimates, standard errors and confidence limits for the beta parameters. If you want to do more than look at the estimates, the function result can be assigned to an object to manipulate the values:

```
> mycoef=coef(myfirstmodel)
> mycoef$estimate
[1] 0.2421484 2.2262658
> mycoef$se/mycoef$estimate
[1] 0.4212817 0.1460335
```

Likewise, the value returned by summary(myfirstmodel) is a list which has class="summary.mark" so if it is not assigned to another object, it is simply printed (by print.summary.mark). However, it is also possible to assign it to another object as shown below:

```
> mysummary=summary(myfirstmodel)

> mode(mysummary)
[1] "list"

> class(mysummary)
[1] "summary.mark"

> names(mysummary)
 [1] "model"      "title"      "model.name" "model.call" "npar"
 [6] "lnl"        "AICc"       "beta"       "reals"      "brief"

> mysummary$model
[1] "CJS"

> mysummary$npar
[1] 2

> mysummary$lnl
[1] 666.8377

> mysummary$beta
                 estimate        se       lcl       ucl
Phi:(Intercept) 0.2421484 0.1020127 0.0422035 0.4420933
p:(Intercept)   2.2262658 0.3251093 1.5890516 2.8634801
```

First Lab Exercise: Simple analysis of the Dipper data

1) *Run the default dipper analysis as shown in the lecture and save it to an object which you can name to your liking.*
2) *Find the help file for* `summary.mark` *and read about the various arguments you can set.*
3) *Produce a summary of your model with standard errors and confidence intervals (Hint: you'll need to set one of the arguments in* `summary`*).*
4) *Extract the Phi parameters. Why are there so many values? Set another argument in* `summary` *to get only the unique values.*
5) *Compute AICc from the log-likelihood, number of parameters and the sample size. The formula is:* $AICc = -2\log(\mathcal{L}) + 2K\left(\frac{n}{n-K-1}\right)$ *where K is the number of parameters and n is the sample size. You'll want to look closely at the help file (*`?mark`*) for the definitions of the values in result.*
6) *Look at the input file that RMark constructed using the* `file.edit` *function. Hint: you can construct the input filename as* `paste(myfirstmodel$output,".inp",sep="")` *where* `myfirstmodel` *is replaced with the name you chose for your model object.*

V. How do I import data into R for RMark?

a. *RMark format: import.chdata*

Previously we discussed what kind of data are in a dataframe for RMark but we have not discussed how to get data into a dataframe for RMark. You have many different options with R to create the dataframe but we are going to discuss two approaches with functions in RMark that were written specifically for the task: `import.chdata` and `convert.inp`. The latter is only useful if you are using data that have already been set up for MARK, so we will start with the first.

You need to start with the data in a tab-delimited text file. You can use a text editor and separate each field with a tab or create with Excel or Access, by choosing the File Save As or Export in Access and select the Save As Type: Text (Tab delimited)(*.txt). The only thing you have to be careful about with Excel is to make sure that the capture history field is a character string. If you enter the values into Excel make sure to put a single quote mark before the string (e.g., '001001). If you use the concatenate function in Excel to create the capture history then it will be a string by default. If for some reason you read the .txt file back into Excel make sure that use the "Data/Text to columns" and define the ch field as text so it stays as character. If you find that all the leading 0s are gone then you have inadvertently converted it to numeric which you do not want. Before converting to a text file, add column labels above each field and use an underscore or period instead of blanks in names. The capture history field must be named "`ch`" in lowercase. Alternatively, you can add names to the fields with `import.chdata`.

Let's look at the example file "dipper.txt". This is the original dipper data with an added `weight` field that I created to make the example more interesting. So the file has 3

fields: ch, sex and weight. The first 2 are character data and weight is numeric. We want to be able to use sex as a factor to create groups and we want to use weight as an individual covariate as a predictor for survival. We can import the data as newdipper as follows:

```
> newdipper=import.chdata("dipper.txt",field.types=c("f","n"))
```

A common mistake is to forget to assign it to an object (e.g., newdipper) in which case the data are imported but they simply fly across the screen and are not saved. The first field must always be "ch" which must always be character data, so it is not necessary to specify it in the argument field.types. But for the other 2 we specify a vector with "f" for factor and "n" for numeric. If you do a summary on newdipper, we see that each field has the type we want:

```
> summary(newdipper)
      ch               sex          weight
 Length:294         Female:153   Min.   : 1.69
 Class :character   Male  :141   1st Qu.: 8.03
 Mode  :character                Median :10.33
                                 Mean   :10.20
                                 3rd Qu.:12.46
                                 Max.   :18.50
```

The default for field.types is "f" for each field, so if we forgot the definition for field.types we would get:

```
> newdipper=import.chdata("dipper.txt")
> summary(newdipper)
      ch               sex          weight
 Length:294         Female:153   10.64  :  3
 Class :character   Male  :141   11.57  :  3
 Mode  :character                12.54  :  3
                                 7.79   :  3
                                 8.03   :  3
                                 10.08  :  2
                                 (Other):277
```

Notice the difference in the summary for weight, which is now shown like a factor with the frequency of an incomplete list of the different weight values. That is not what we want because RMark will not allow use of a factor variable as an individual covariate.

The function import.chdata is simply a convenience function to import data for RMark but it is not the only way to create a dataframe for RMark. You can use any of the tools in R to create a dataframe; however, you do need to meet the requirements for the data structure that I described previously.

b. MARK format: convert.inp

If you already have an ".inp" file that you have used with MARK, use the function convert.inp to create the dataframe for RMark. As an example we will use the dipper.inp file that is in the MARK program's Examples subdirectory. The data format for a MARK input file is quite different than the RMark format. In particular, each record (row/entry) in a MARK input file can represent animals from different groups and a

frequency for each group is specified for each record. Also, each record must end with a semicolon. To convert the input file, you must describe the values of the factor variables that should be used for the groups in the order they are given in each record of the input file. The structure for the dipper.inp file is fairly simple with 2 groups: males followed by females. So this file can be converted as follows:

```
> dipper=convert.inp("dipper",group.df=data.frame(sex=c("Male","Female")))
> summary(dipper)
      ch                freq       sex
 Length:294         Min.   :1   Female:153
 Class :character   1st Qu.:1   Male  :141
 Mode  :character   Median :1
                    Mean   :1
                    3rd Qu.:1
                    Max.   :1
```

The arguments are `inp.filename` (name of the file) which is "dipper.inp" (.inp is assumed) for this example and `group.df` which is a dataframe that defines the name(s) of the variables used to define the factor variables and the levels of the variables in the order in the file. Above I chose to use "sex" as the factor variable name and to call the levels "Male" and "Female". I could change the names/values but not the order which was specified by the file. A much more complicated example with two different factor variables is described in Appendix C.12 with the multi_group.inp file which is available with the Cooch and White electronic book.

c. *Exporting data for MARK; export.chdata*

If your data are in RMark but you want to set them up in MARK to do something like median c-hat, you need to export the data into an ".inp" format using the function `export.chdata`. This is jumping ahead slightly to the next section, so without explanation I'll show how this function is used with the newdipper dataframe we created to create a file named newdipper.inp.

```
> newdipper.proc=process.data(newdipper,model="CJS",groups="sex")
> export.chdata(newdipper.proc,"newdipper",covariates="weight")
```

Now, we can use `convert.inp` and store in another object named `newdipper2` and see that the first 5 records are identical after the conversion process.

```
> newdipper2=convert.inp("newdipper",
     group.df=data.frame(sex=c("Female","Male")),covariates="weight")

> head(newdipper,5)
       ch    sex weight
1 0000001 Female   9.09
2 0000001 Female   6.16
3 0000001 Female  10.73
4 0000001 Female  13.82
5 0000001 Female  13.59

> head(newdipper2,5)
         ch freq    sex weight
1:1 0000001    1 Female   9.09
1:2 0000001    1 Female   6.16
1:3 0000001    1 Female  10.73
```

```
1:4 0000001    1 Female  13.82
1:5 0000001    1 Female  13.59
```

With `convert.inp` you have to provide `group.df` which lists the covariates that will be assigned to the groups as they are defined and ordered in the .inp file. In this case they are ordered with "Female" first and then "Male" because the .inp file was created with `export.chdata` using sex as a group factor variable and R creates factor levels in alphabetical order (F before M). This will become clearer when we describe how to create groups with the `process.data` function which we will discuss next.

VI. A more realistic overview example

a. Describing your model and data to RMark: process.data

So far we have only created the default model (Phi(.)p(.)) for the default type of model (CJS). Next you need to learn the steps needed for more useful analysis of capture-recapture data. Although, you can use the `mark` function to do most everything it is more efficient and more flexible to do two of the steps in `mark` prior to calling `mark` rather than having them done each time `mark` is invoked.

The first step is to use the function `process.data` which creates a processed data list that sets various values such as the type of model (e.g., CJS, POPAN), time intervals and beginning time, group variables and others. Using this function is similar to the process of creating a project with the first screen in the MARK GUI interface. The processed data list it creates is the primary data object that is used by RMark because it contains all of the necessary descriptors for the data. For example, consider the call to `process.data` that I used above:

```
> newdipper.proc=process.data(newdipper,model="CJS",groups="sex")
```

I use the suffix ".proc" to differentiate between the original dataframe and the processed data list created by `process.data`. You can see that I specified the type of model would be CJS and I wanted to use "sex" as a grouping variable to use it to explore differences in survival or capture probability between the sexes. I could have also set a beginning time for the first capture (release) occasion which is used for labeling both beta and real parameters:

```
> newdipper.proc=process.data(newdipper,model="CJS",
                              groups="sex",begin.time=1990)
```

Except for model="Nest", there is no need to specify the number of occasions because this is determined from the length of the capture history. Also, if the capture history has any values that are inappropriate for the type of model or the capture histories are not of the same length then an error will be issued when you use `process.data`. For example, if I deliberately change the lengths of the capture histories for rows 1 and 10, the function will report that for those rows the capture history lengths are different:

```
> data(dipper)
> baddipper=dipper
> baddipper$ch[1]="001"
> baddipper$ch[10]="001"
> dipper.proc=process.data(baddipper,model="CJS",groups="sex")
Error in process.data(baddipper, model = "CJS", groups = "sex") :
```

```
Capture history length is not constant. ch must be a character string
 row numbers with incorrect ch length 1,10
```

Likewise, if I add a character other than 0 or 1, it also complains by listing the set of values used in the capture histories. You need to know which are valid:

```
> data(dipper)
> baddipper=dipper
> baddipper$ch[1]="0020101"
> dipper.proc=process.data(baddipper,model="CJS",groups="sex")
Error in process.data(baddipper, model = "CJS", groups = "sex") :
Incorrect ch values in data:021
```

I will explain other arguments for `process.data` as we encounter a need for them. Don't forget ?process.data will give an explanation for all the arguments and examples.

b. *Creating design "data" for the models: make.design.data*

The next step is even easier. Earlier I mentioned the concept of design "data" in describing PIMs and their relationship to model structure. The function `make.design.data` uses the processed data list which results from `process.data` and it creates a default set of design data that are appropriate for the data and type of model. I use the suffix ".ddl" for the design data list that results from the function call:

```
> data(dipper)
> dipper.proc=process.data(dipper,model="CJS",groups="sex",
                                          begin.time=1990)
> dipper.ddl=make.design.data(dipper.proc)
```

There are other arguments for this function but they are not typically needed. We will get into details about the design data later but for the time being I just want to show you that the design data is a list with a dataframe for each parameter and another entry called pimtypes which I'll discuss later as well.

```
> mode(dipper.ddl)
[1] "list"
> names(dipper.ddl)
[1] "Phi"      "p"        "pimtypes"
```

If we do a summary of the design data for Phi we see that it created the factor variables group, cohort, age, time and sex and it created the numeric variables Cohort, Age and Time. The use of a capital first letter is a convention to distinguish numeric from factor in the default design data.

```
> summary(dipper.ddl$Phi)
    group     cohort   age      time        Cohort           Age
 Female:21   1990:12   0:12   1990: 2   Min.   :0.000   Min.   :0.000
 Male  :21   1991:10   1:10   1991: 4   1st Qu.:0.000   1st Qu.:0.000
             1992: 8   2: 8   1992: 6   Median :1.000   Median :1.000
             1993: 6   3: 6   1993: 8   Mean   :1.667   Mean   :1.667
             1994: 4   4: 4   1994:10   3rd Qu.:3.000   3rd Qu.:3.000
             1995: 2   5: 2   1995:12   Max.   :5.000   Max.   :5.000
      Time           sex
 Min.   :0.000   Female:21
 1st Qu.:2.000   Male  :21
 Median :4.000
 Mean   :3.333
 3rd Qu.:5.000
```

```
 Max.   :5.000
```

Always keep in mind that design data describe the model structure and are data describing model parameters and are not data about specific animals. The concept of design data is what made it possible for RMark to create models for MARK. As you'll see later, the design data provide a great deal of flexibility for the modeling.

c. *Specifying formulas for the model*

Now we can get to the meat of RMark which is to specify formulas to create models rather than having to create all those #!$#^ design matrices by hand. I will use newdipper which we created earlier for these examples. I won't go into a lot of depth at this point because I just want you to see how it is done at this point. But you do need to understand how to construct an R formula as it is the basis of model specification in RMark.

Let's step back and think about an algebraic equation for a line y=a+b*x where y is the dependent variable, x is the independent variable and the parameters are intercept (a) and slope (b). In an R formula, you only specify the variables used in the equation and not the parameters. So for a linear regression the equation would be specified as y~x where the ~ is analogous to =. It is "understood" that there is an intercept and if x is a numeric variable then the other parameter is the slope for the x. If you wanted the model y=bx with no intercept, then in R that would be y~-1+x where the -1 means to remove the intercept. If you wanted a horizontal line (y=a) with no variable x, then you would use y~1. Now in some cases, the dependent variable is implicit and is not specified, so y is dropped.

With RMark we will specify a formula for each of the parameters and we can use any of the variables in the design data and any numeric individual covariates. So if we wanted a formula with constant Phi or p we would use ~1. That is the default formula for Phi and p for CJS. If we wanted them to vary for each occasion we could use ~time or if we wanted a trend over time we could use the numeric equivalent ~Time. If we thought that the dipper's weight at initial capture would affect its future survival and survival varied by sex we could create an additive formula of ~weight + sex.

More on putting together formulae later but now you need to know how to specify these for the parameters. You specify a model with RMark by specifying a formula for each parameter. Each parameter specification is a list that contains a formula (at the very least), and possibly other values like fixed parameter values, etc. It is best to create a separate R object for each parameter specification and give it a recognizable or explanatory name. For reasons that will be obvious later, you'll want to use a prefix that is the parameter name followed by a period and then any other part of the name after the period. For example,

```
Phi.dot=list(formula=~1)
p.time=list(formula=~time)
Phi.weight.sex=list(formula=~weight+sex)
```

If you were to examine Phi.weight.sex you would see that it is a list with one element named formula with the value ~weight+sex:

```
> Phi.weight.sex
$formula
~weight + sex
```

So how do we use these formulae to specify different models using newdipper? First, we need to process the data and create the design data:

```
> newdipper.proc=process.data(newdipper,model="CJS",begin.time=1990,groups="sex")
> newdipper.ddl=make.design.data(newdipper.proc)
```

Next we need to use the `mark` function with the first 2 arguments specifying the processed data list and the design data list and we add `model.parameters` which specifies a list with the specification list for each parameter:

```
> model.1=mark(newdipper.proc,newdipper.ddl,
               model.parameters=list(Phi=Phi.dot,p=p.time))
> model.2=mark(newdipper.proc,newdipper.ddl,
               model.parameters=list(Phi=Phi.weight.sex,p=p.time))
Error in make.mark.model(data.proc, title = title, covariates = covariates,  :
  The following individual covariates are not allowed because they are factor
variables:  weight
```

So what happened? Ah yes, remember how I demonstrated the incorrect way to enter the data such that weight became a factor variable? So how do I fix that? First, I need to re-enter the data properly:

```
> newdipper=import.chdata("dipper.txt",field.types=c("f","n"))
```

But, that only changed newdipper and not newdipper.proc which contains the processed data, so I have to repeat that call:

```
> newdipper.proc=process.data(newdipper,model="CJS",
                                    begin.time=1990,groups="sex")
```

Now because I didn't change the model structure I don't need to change the design data but as you'll see later when you use scripts, it is often easier to run everything again. Now, the model should run fine:

```
> model.2=mark(newdipper.proc,newdipper.ddl,
                 model.parameters=list(Phi=Phi.weight.sex,p=p.time))
```

d. Computing real parameter estimates with individual covariates

When you use an individual covariate, the real parameters depend on the value of the covariate (e.g., survival depends on weight). The real parameter values you see in the summary output use the average of the covariate values in the data for the calculation of the real parameters. However, you can easily obtain real parameter estimates at a series of covariate values using the function `covariate.predictions`. Without a whole lot of explanation at present, I'll show how you can get 20 estimates of female survival at weight values from 1 to 20 for parameter index 1 (`index=rep(1,20)`):

```
Phi.weight.predictions=covariate.predictions(model.2,
```

```
                                    data=data.frame(index=rep(1,20),weight=1:20))
```

The function returns a list with a dataframe named `estimates` and a matrix named `vcv` which is the variance-covariance matrix for the real parameters. We'll use the estimates and their confidence intervals below to produce a plot of the weight-Phi relationship.

e. *Plotting real parameter values*
As with most aspects of R, there are many different ways to produce graphics and the range of options expands continually. It is not my area of expertise, so there may be a better way to do plots than what I'm going to show you. With that caveat in mind, the following will plot the weight-Phi relationship with pointwise confidence intervals:

```
> with(Phi.weight.predictions$estimates,
{
    plot(weight, estimate,type="l",lwd=2,xlab="Weight(g)",
                                  ylab="Survival",ylim=c(0,1))
    lines(weight,lcl,lty=2)
    lines(weight,ucl,lty=2)
})
```

f. *Cleaning up - removing unused files*
If you have been creating models and not saving them or deleting them afterwards, it is a good idea to tidy up a bit and throw out the "garbage" with a function called `cleanup`. As I described earlier, files are created in your directory by MARK and these are linked to R objects if you save them. However, if you don't save an R object or you delete it later, the MARK output files remain behind. Now they aren't hurting anything but they are taking up disk space so best to remove them. You can do that by typing:

```
> cleanup(ask=FALSE)
```

It looks through your workspace (.Rdata) and identifies each R object of class "mark" and creates a list of all the marknnn.* files that are in use and compares that to the list in your directory. Any files that are not linked to an R object are deleted from your directory. The argument ask=FALSE simply makes it clear that it will do it without asking permission for each file.

Second Lab Exercise: importing/exporting data; specifying models with formulas

1) *Import the dipper.txt file into a dataframe using import.chdata*
2) *Use export.chdata to create an input file for MARK with no sex grouping*
3) *Use file.edit() to examine the .inp file you created. What is in the file?*
4) *With the dataframe constructed from dipper.txt, create the following models:*
    a. *Phi=~1, p=~1*
    b. *Phi=~1, p=~time*
    c. *Phi=~1, p=~Time*
    d. *Phi=~sex, p=~1*
    e. *Phi=~sex, p=~time*
    f. *Phi=~sex, p=~Time*
    g. *Phi=~sex+weight, p=~1*

h. *Phi=~sex+weight, p=~time*
i. *Phi=~sex+weight, p=~Time*

5) *Which model is the most parsimonious – smallest AICc value?*

VII. Organized workflow

a. *Creating a script/function to analyze data*

You may have already discovered in the exercises that it is quite easy to make mistakes by forgetting a comma or parenthesis here and there. Also, hopefully with the last exercise it was obvious that if you are going to have many different models there would be lots of redundant typing. Also, you would begin to have various formula and model objects cluttering your workspace to the point of not being able to keep it all organized. As I developed RMark this became obvious to me so I produced code to help organize the workflow of building models.

The first step in organizing any workflow in R is to recognize the value of scripts and functions and to use them even with trivial projects. First of all what is a script? A script is simply a text file of R code that you can use in R with either the R script editor (File/New script) or with an external text editor like TINN-R. With an external editor you can use 1) File/Source R code on the menu or with the source function, 2) copying and pasting text, or 3) built-in transmission of text to R with TINN-R like the R script editor. One of the advantages of using TINN-R is the syntax highlighting that helps with matching parentheses and braces etc. With a script you can add comments (#blah blah blah) ad nauseum to document what you have done and why. Doing so can be a big time saver if you have to pick up an analysis 6 months after you did it when some reviewer asks a question about the specifics of the analysis you did for your paper. Also, if you include the graphics for your figures in your script, you can easily change the font or title when the editor or publisher wants you to make a change. But even more importantly, the script provides the mechanism to replicate fully the analysis you did on a particular data set. If there is any question about what was done or how you can always go back to the script and the data that it used. If you find a mistake, it can be corrected and the analysis can be run again with the correction.

Scripts can create functions but functions are fundamentally different in that the code inside of a function operates within the function scope. For example, if you use `ls()` to get a list of objects at the command line, it will show you all the objects currently in your workspace. However, if you use `ls()` in the function `myf` defined below, then it will only show the object x defined within the function:

```
ls()
myf=function()
{
  x=1
  ls()
}
myf()
```

If you were to refer to `x` in the function it would use the `x` in that function and not any `x` that was defined outside of `myf`. But if it uses an object `y` that doesn't exist in `myf` that

exists in the parent environment (typically the workspace) of `myf`, it will use that object. In other words, if you create an object in a script (e.g., processed data list) you can use that object in the function that you also create in the script without passing it as an argument. If you don't understand some of this, it is no big loss. The main point is that you'll want to use both scripts and functions to organize and document your workflow and I'll demonstrate a template you can follow.

b. Creating all combinations of models from sets of parameter sub-models

From exercise 2, it should be obvious that even RMark could become a pain to create a set of 100 models with all combinations of 10 models for Phi and 10 models for p. That would mean writing out 100 calls to mark after creating the 20 parameter specifications. Not something that I want to do! Which is why I created a couple of functions called `create.model.list` and `mark.wrapper` that should be used within a script and function that you will write to analyze your data. First, I'll create a script that will repeat the analysis we did in Exercise 2 for the newdipper data. You can find it in myscript.r.

```
###  myscript.r ###
# Import the newdipper data from dipper.txt. It has fields ch, sex and weight
newdipper=import.chdata("dipper.txt",field.types=c("f","n"))
# Process the dataframe with the CJS model and use sex to define 2 groups
newdipper.proc=process.data(newdipper,model="CJS",begin.time=1990,groups="sex")
# Create the default design data
newdipper.ddl=make.design.data(newdipper.proc)
# Create a function called do.analysis that will create all the models that I want
do.analysis=function()
{
# Within the function define the set of formulae that I want for Phi and p
Phi.dot=list(formula=~1)
Phi.sex=list(formula=~sex)
Phi.sex.weight=list(formula=~sex+weight)
p.dot=list(formula=~1)
p.time=list(formula=~time)
p.Time=list(formula=~Time)
# Use create.model.list to construct each of the 9 combinations of the Phi-p models
# and store it in the object named cml.  The argument is the type of model, CJS in
# this example.  Based on that argument it looks for objects created in this
# function that are lists containing a formula element that have names starting
# with the parameter name followed by a period (e.g., Phi. and p. for CJS). It
# ignores any similar objects defined elsewhere which is the primary reason to
# use a function to do this.
cml=create.model.list("CJS")
# Finally use mark.wrapper with this model list and the processed data and design
# data to fit each of the models with mark
model.list=mark.wrapper(cml,data=newdipper.proc,ddl=newdipper.ddl)
# Return the list of model results as the value of the function
return(model.list)
}
# The above only created the function do.analysis.  To run it, I have to assign it
# to an object as follows:
myresults=do.analysis()
# show the results
myresults
###  end of myscript.r ###
```

If you look at this simple script broadly there are 4 tasks and the first 2 you have already encountered with processing the data and creating the design data. The third task is to write a function with the set of parameter specifications that you want to examine and to call `create.model.list` and `mark.wrapper` and return the results from the latter. The final task is to call the function you created to fit the models and store the results

which we will describe next. Scripts can become more complex once you begin to manipulate design data, produce plots or model average values, etc. However, the basic structure outlined above can always be used as the initial template for an analysis script. Just make sure to document, document, document!

c. Model selection and working with a marklist

So just what was created by the function `mark.wrapper` and stored in `myresults` in the script? It is a `marklist` which is a list with a particular structure that has a class of "marklist". A `marklist` contains a list element for each model that was run and an element called `model.table` which is a summary table of model results that is similar to the table that you see in the MARK GUI. If you type the name of a `marklist`, it will print a summary of the `model.table`:

```
> mode(myresults)
[1] "list"
> class(myresults)
[1] "marklist"
> names(myresults)
 [1] "Phi.dot.p.dot"          "Phi.dot.p.time"
 [3] "Phi.dot.p.Time"         "Phi.sex.p.dot"
 [5] "Phi.sex.p.time"         "Phi.sex.p.Time"
 [7] "Phi.sex.weight.p.dot"   "Phi.sex.weight.p.time"
 [9] "Phi.sex.weight.p.Time" "model.table"

> myresults
                        model npar     AICc DeltaAICc      weight  Deviance
3            Phi(~1)p(~Time)    3 670.8170   0.00000 0.256661212  82.28302
1               Phi(~1)p(~1)    2 670.8660   0.04903 0.250445661  84.36055
9 Phi(~sex + weight)p(~Time)    5 671.9775   1.16053 0.143666117 661.83468
7    Phi(~sex + weight)p(~1)    4 672.0216   1.20456 0.140537868 663.92654
6          Phi(~sex)p(~Time)    4 672.6852   1.86818 0.100853299  82.11306
4             Phi(~sex)p(~1)    3 672.7331   1.91608 0.098466558  84.19909
2            Phi(~1)p(~time)    7 678.7481   7.93111 0.004865658  82.00306
8 Phi(~sex + weight)p(~time)    9 679.9837   9.16667 0.002623271 661.55098
5          Phi(~sex)p(~time)    8 680.6496   9.83259 0.001880358  81.82716
```

The number on the left of the `model.table` is the model number and represents its position in the `marklist`. Thus, if you want to work specifically with the best model, you could refer to it by `myresults[[3]]` in this example. The `model.table` is listed in increasing order of DeltaAICc (i.e., best to worst model). It shows the model name, the number of parameters (npar), the AICc, DeltaAICc, model weight and deviance. The latter was not a good default to include in the table because it is not comparable across models with and without individual covariates. A better choice would have been to show -2* log likelihood which can be shown if you set the argument `use.lnl`=TRUE in the function `model.table` as shown below:

```
> myresults$model.table=model.table(myresults,use.lnl=TRUE)
> myresults
                        model npar     AICc DeltaAICc      weight  Neg2LnL
3            Phi(~1)p(~Time)    3 670.8170   0.00000 0.256661212 664.7601
1               Phi(~1)p(~1)    2 670.8660   0.04903 0.250445661 666.8377
9 Phi(~sex + weight)p(~Time)    5 671.9775   1.16053 0.143666117 661.8347
7    Phi(~sex + weight)p(~1)    4 672.0216   1.20456 0.140537868 663.9265
6          Phi(~sex)p(~Time)    4 672.6852   1.86818 0.100853299 664.5902
4             Phi(~sex)p(~1)    3 672.7331   1.91608 0.098466558 666.6762
```

```
2            Phi(~1)p(~time)    7 678.7481   7.93111 0.004865658 664.4802
8 Phi(~sex + weight)p(~time)    9 679.9837   9.16667 0.002623271 661.5510
5          Phi(~sex)p(~time)    8 680.6496   9.83259 0.001880358 664.3043
```

d. Model averaging

Often there is no clear best model in an analysis and one solution is to compute a weighted average of the parameters across the set of models. The model weights are the AICc derived weights shown in the `model.table`. Model averaged parameter estimates can be obtained with the function `model.average` and `covariate.predictions`. If you have individual covariates in any of the models for a parameter, you'll want to use `covariate.predictions` for that parameter; otherwise use `model.average`.

So for this example, we will use `model.average` for p and for Phi we will use `covariate.predictions` because it had weight as an individual covariate. If you use `model.average` and do not request computation of a variance-covariance matrix (`vcv=TRUE`), it will return a dataframe with the estimates and standard errors:

```
> mavg.p=model.average(myresults,"p")
> head(mavg.p)
                      par.index  estimate        se fixed    note  group cohort age time Cohort Age Time    sex
p gFemale c1990 a1 t1991        43 0.8523922 0.08198948              Female   1990   1 1991      0   1    0 Female
p gFemale c1990 a2 t1992        44 0.8762532 0.04989749              Female   1990   2 1992      0   2    1 Female
p gFemale c1990 a3 t1993        45 0.8943980 0.03233340              Female   1990   3 1993      0   3    2 Female
p gFemale c1990 a4 t1994        46 0.9089871 0.02956691              Female   1990   4 1994      0   4    3 Female
p gFemale c1990 a5 t1995        47 0.9202411 0.03426209              Female   1990   5 1995      0   5    4 Female
p gFemale c1990 a6 t1996        48 0.9287401 0.03989160              Female   1990   6 1996      0   6    5 Female
```

Each line above is one of the model averaged parameter estimates for p. It provides the estimate, standard error (se) and some information about the group, age and cohort that the parameter represents. If a real parameter value has been fixed, the word "fixed" will appear in the column fixed. The column note is a carryover and may show a note that the interval had been computed as a profile interval in the MARK run.

If you set `vcv=TRUE`, you'll get a list with elements 1) `estimates`: a dataframe with the estimates including confidence intervals, and 2) `vcv.real`: a variance-covariance matrix for the real parameters:

```
> mavg.p=model.average(myresults,"p",vcv=TRUE)
> names(mavg.p)
[1] "estimates" "vcv.real"
> head(mavg.p$estimates[,1:5])
                      par.index  estimate         se       lcl       ucl
p gFemale c1990 a1 t1991        43 0.8523922 0.08198948 0.6168664 0.9539422
p gFemale c1990 a2 t1992        44 0.8762532 0.04989749 0.7418279 0.9457996
p gFemale c1990 a3 t1993        45 0.8943980 0.03233340 0.8123726 0.9430768
p gFemale c1990 a4 t1994        46 0.9089871 0.02956691 0.8321358 0.9526562
p gFemale c1990 a5 t1995        47 0.9202411 0.03426209 0.8221079 0.9664490
p gFemale c1990 a6 t1996        48 0.9287401 0.03989160 0.7999676 0.9769979
```

If you use individual covariates in a model, model averaging gets a little more complicated because you have to decide what real parameter estimates you want to compute. Recognize that there are potentially an infinite number of estimates you could compute. Fortunately, there are typically some obvious choices of either using averages of the covariate or computing the values at a finite set of covariate values to display the relationship between the real parameter and the covariate, like we did in an earlier

example. By default, the real estimates are computed using the average covariate value but this is done based on the average across the entire data set. For the dipper example, it would make sense to compute the real parameter values using the average weight for each cohort (release year) for each sex. We can use the `tapply` function to get the mean weights for each of the 12 classes (2 sexes * 6 cohorts) but first we need a way to split the data into cohorts. The following is a useful little function that creates a cohort factor variable that will work with any single position capture history. It would have to be modified to work with live-dead (LD) format which uses 2 positions in the ch for each occasion:

```
create.cohort=function(x)
{
# split the capture histories into a list with each list element
# being a vector of the occasion values (0/1). 1001 becomes
# "1","0","0","1"
   split.ch=sapply(x$ch,strsplit,split="")
# combine these all into a matrix representation for the ch
# rows are animals and columns are occasions
   chmat=do.call("rbind",split.ch)
# use the defined function on the rows (apply(chmat,1…) of the
# matrix.  The defined function figures out the column containing
# the first 1 (its initial release column).
   return(factor(apply(chmat,1,function(x) min(which(x!="0")))))
}
```

We can use it to create a cohort variable in `newdipper` as follows:

```
> newdipper$cohort=create.cohort(newdipper)
> summary(newdipper)
      ch                sex          weight      cohort
 Length:294         Female:153   Min.   : 1.69   1:22
 Class :character   Male  :141   1st Qu.: 8.03   2:49
 Mode  :character                Median :10.33   3:52
                                 Mean   :10.20   4:45
                                 3rd Qu.:12.46   5:41
                                 Max.   :18.50   6:46
                                                 7:39
```

Now that newdipper contains the cohort variable we can get the mean weights:

```
> mean.wts=with(newdipper, tapply(weight, list(cohort,sex), mean))
> mean.wts
     Female      Male
1 10.935000  9.735833
2 10.361724 10.417500
3  8.929259 10.449600
4 10.427391 11.368636
5 10.661053 10.158636
6  9.508261 10.510870
7  8.692727 11.443529
```

We don't want the values for cohort 7 which are not used in the analysis and we want the weights to be represented as a vector in `covariate.predictions`. The R code

below will use the `as.vector` function to change the first 6 rows of the `mean.wts` to a vector with 12 values:

```
> mean.wts=as.vector(mean.wts[1:6,])
> mean.wts
 [1] 10.935000 10.361724  8.929259 10.427391 10.661053  9.508261
9.735833 10.417500 10.449600 11.368636 10.158636 10.510870
```

Did you see that they are ordered by going from top to bottom and left to right? This is the standard approach that R uses in converting a matrix or array to a vector or in filling a matrix/array. You can change the order by using the transpose function (`t(x)`) :

```
 > mean.wts=with(newdipper, tapply(weight, list(cohort,sex), mean))
> mean.wts=as.vector(t(mean.wts[1:6,]))
> mean.wts
 [1] 10.935000  9.735833 10.361724 10.417500  8.929259 10.449600
10.427391 11.368636 10.661053 10.158636  9.508261 10.510870
```

Another way to switch the order would be to switch the order of cohort and sex in the list for `tapply`. Either order will work but it is slightly more convenient for this application to have all the weights for females to be followed by males but we'll use them as they are listed above so you can learn some more R. We'll want to link the average weights to specific Phi parameters for `covariate.predictions`. For this example, we only have sex and weight effects for Phi, so there are no differences by time or by cohort. Therefore, we can use a single Phi for females and a single Phi for males. At present, without telling you how I got the values, we can use the numbers (indices) 1 and 22 for females and males, respectively. The following will get the predictions that we want:

```
> Phi.by.wt=covariate.predictions(myresults,
             data=data.frame(index=rep(c(1,22),6),weight=mean.wts))
```

Take note of a few things about the creation of the dataframe for the data argument. First, `index` is constructed by repeating the vector `c(1,22)` which alternates the female and male indices to match the order in the `mean.wts` vector. Had we used the vector constructed in the other order, we would have used `index=rep(c(1,22), each=6)`or `index=c(rep(1,6),rep(22,6))`which both create the vector `c(1,1,1,1,1,1,22,22,22,22,22,22)`. Second, `mean.wts` was stored in the dataframe with the name `weight` because that is its name in the data used to fit the model. The name given in the dataframe (argument `data`) for the covariate must match the name (e.g., weight) for the individual covariate in the original dataframe (e.g., `newdipper`).

```
> names(Phi.by.wt)
[1] "estimates" "vcv"
> Phi.by.wt$estimates
   vcv.index model.index par.index index    weight  estimate         se       lcl       ucl fixed
1          1           1         1     1 10.935000 0.5559409 0.03009538 0.4964415 0.6138781
2          1           1        22    22  9.735833 0.5592285 0.03054840 0.4987809 0.6179697
3          1           1         1     1 10.361724 0.5537666 0.02997077 0.4945456 0.6114998
4          1           1        22    22 10.417500 0.5618066 0.03042477 0.5015665 0.6202782
5          1           1         1     1  8.929259 0.5483059 0.03188768 0.4853725 0.6097313
6          1           1        22    22 10.449600 0.5619278 0.03043682 0.5016618 0.6204201
7          1           1         1     1 10.427391 0.5540161 0.02995841 0.4948160 0.6117225
8          1           1        22    22 11.368636 0.5653830 0.03143128 0.5030825 0.6256844
9          1           1         1     1 10.661053 0.5549028 0.02997064 0.4956658 0.6126198
10         1           1        22    22 10.158636 0.5608289 0.03038630 0.5006799 0.6192426
```

```
11        1          1        1     1  9.508261 0.5505173 0.03075197 0.4897962 0.6097694
12        1          1       22    22 10.510870 0.5621589 0.03046425 0.5018348 0.6206992
```

Now, maybe an obvious thing to do would be to plot the estimates with error bars for each cohort. To do that I'll use a function in the Hmisc package called `errbar`. Remember my caveat about my R plotting knowledge! I use what I know.

```
> par(mfrow=c(2,1))
> library(Hmisc)
> ?errbar
> with(Phi.by.wt$estimates,
errbar(1990:1995,estimate[seq(1,12,2)],lcl[seq(1,12,2)],ucl[seq(1,12,2)],
          ylim=c(.4,.8),xlab="Cohort",ylab="Female survival"))
> with(Phi.by.wt$estimates,
errbar(1990:1995,estimate[seq(2,12,2)],lcl[seq(2,12,2)],ucl[seq(2,12,2)],
          ylim=c(.4,.8),xlab="Cohort",ylab=" Male survival"))
```

The seq(1,12,2) creates a vector 1,3,5…,11 and the seq(2,12,2) creates a vector 2,4,6,…12. These vector of indices extract the values for females (odds) from `wt$estimates` and for males (evens) because they ended up alternating in the estimates due to the way that I constructed the call to `covariate.predictions`. The values don't vary substantially as shown in the plot, so if this was a real analysis you might choose to report a single value using the average weight. I haven't given it much thought but there is probably a way to incorporate a random-effects approach in this situation to capture the variability as well.

VIII. Digging into the details of RMark: PIMS and design data

a. PIMs are alive and well (unfortunately)

Even though you don't have to create PIMs with RMark like you do with the MARK interface, they are lurking in the background unfortunately. I say unfortunately because there are ways to formulate capture-recapture models without PIMs but not with MARK. If you are interested read about some of the future developments I'm adding in RMark (?crm and ?create.dmdf). For the time being we are stuck with PIMs because they are used by MARK. That comment is not meant to be a criticism of MARK because it is a truly amazing program, but PIMs are a downright nuisance and they can be extremely confusing for many folks using the software. By relegating PIMS to the background, RMark hopefully reduces the confusion, but you can't completely ignore them. So far I've only made brief mention of these mysterious "indices" like in the last example, but the time has come to open up that can of worms!

If you didn't understand the discussion under IV.a, you should probably go back and read it again before you read this section because this section will expand on the earlier material. The default PIM structure in RMark is the all-different PIM in which each parameter is given a different index (number). It is possible to use either a "time" or "constant" PIM and that can be useful in limited circumstances. I won't discuss that here because it isn't necessary for most, but if you are interested see `?make.design.data` and `?make.mark.model`. You can look at the PIMs that any model is using by extracting the `pims` object from the `mark` object. For example, the PIMs used for p in model 1 from `myresults` are:

```
 > myresults[[1]]$pims$p
[[1]]
[[1]]$pim
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   43   44   45   46   47   48
[2,]    0   49   50   51   52   53
[3,]    0    0   54   55   56   57
[4,]    0    0    0   58   59   60
[5,]    0    0    0    0   61   62
[6,]    0    0    0    0    0   63

[[1]]$group
[1] 1

[[2]]
[[2]]$pim
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   64   65   66   67   68   69
[2,]    0   70   71   72   73   74
[3,]    0    0   75   76   77   78
[4,]    0    0    0   79   80   81
[5,]    0    0    0    0   82   83
[6,]    0    0    0    0    0   84

[[2]]$group
[1] 2
```

The structure is similar for Phi and they are numbered 1 to 21 for Females (group 1) and 22-42 for males (group 2). These indices uniquely identify each possible parameter in the model. Sometimes RMark will also use an index that is not unique across parameter types but is unique within a parameter type. For example, p might be numbered 1-42 as well. In hindsight, this was a REALLY BAD idea. It may be correctable in a future version of RMark but for now you'll have to live with it. Fortunately, you don't use these indices a lot and the reason for this section is simply for you to know what they are, how to use them and to bridge the link between parameters and the design data which are important for model building.

For some types of models, including CJS, you can also view the PIMs with a function named PIMS that I wrote solely to help explain them in situations like this. So to view the same PIMS for p, you could do the following:

```
> PIMS(myresults[[1]],"p",simplified=FALSE)
group = sexFemale
     1991 1992 1993 1994 1995 1996
1990   43   44   45   46   47   48
1991        49   50   51   52   53
1992             54   55   56   57
1993                  58   59   60
1994                       61   62
1995                            63
group = sexMale
     1991 1992 1993 1994 1995 1996
1990   64   65   66   67   68   69
1991        70   71   72   73   74
1992             75   76   77   78
1993                  79   80   81
1994                       82   83
1995                            84
```

The above is slightly easier to read because it removes all the 0's and it labels the groups. Otherwise, it looks like what I did earlier. In the call to `PIMS` I didn't have to use `myresults[[1]]`. I could have used `myresults[[3]]` or any of the model numbers in `myresults` and the output of the PIMS function would look exactly the same because each model is using the same underlying set of all-different PIMS because they are all using the same type of model (CJS), they all have 2 groups and they all have 7 occasions. If you change any one of those, the underlying PIMS will change. Now I have avoided the argument `simplified=FALSE` but we'll get to that later when we discuss design matrices. When we use the defaut of `simplified=TRUE`, then the output of the PIMS function will differ between models.

b. *Viewing model structure as design data*

Each of the parameters in a PIM represents a different quantity. For example, parameter 72 in the above PIMs, is the probability of recapturing in 1994, a surviving male from the cohort originally released in 1991. If all animals were released at age 0, then the age of those males in 1994 would be 3 at the time of recapture. Similarly, parameter 59 is the probability of recapturing in 1995, a surviving 2 year old female from the cohort originally released in 1993. With each parameter we can identify data that we can associate with it to "describe" what that parameter represents in the model. For good or bad, I chose to call these design data because they depend on the design (structure) of the model. A design data list (ddl) is simply a list of the design dataframes for each type of parameter (e.g., Phi, p) in the model. For each type of model (eg CJS), the `make.design.data` function creates design data for each parameter that are appropriate for that parameter of that type of model. Put another way, the design data vary for the parameter and type of model and RMark creates some default data that can be used. You can add to the design data and even modify or delete design data. But, beware because you really need to know what you are doing because these data determine the form of the design matrix from the formula and you can mess up! Just like your capture history data, the model is only useful if the design data are correct. Hopefully, you'll know what not to do after reading this and the remaining sections.

I'll use the same `newdipper` data with the CJS model to create the examples. The design data for a parameter is a dataframe that has its rows in the numeric order of the parameter index.

```
> PIMS(myresults[[1]],"Phi",simplified=F)
group = sexFemale
     1990 1991 1992 1993 1994 1995
1990    1    2    3    4    5    6
1991         7    8    9   10   11
1992             12   13   14   15
1993                  16   17   18
1994                       19   20
1995                            21
group = sexMale
     1990 1991 1992 1993 1994 1995
1990   22   23   24   25   26   27
1991        28   29   30   31   32
1992             33   34   35   36
1993                  37   38   39
1994                       40   41
1995                            42
```

```
> newdipper.ddl$Phi
    group cohort age time Cohort Age Time    sex
1  Female   1990   0 1990      0   0    0 Female
2  Female   1990   1 1991      0   1    1 Female
3  Female   1990   2 1992      0   2    2 Female
4  Female   1990   3 1993      0   3    3 Female
5  Female   1990   4 1994      0   4    4 Female
6  Female   1990   5 1995      0   5    5 Female
7  Female   1991   0 1991      1   0    1 Female
8  Female   1991   1 1992      1   1    2 Female
9  Female   1991   2 1993      1   2    3 Female
10 Female   1991   3 1994      1   3    4 Female
11 Female   1991   4 1995      1   4    5 Female
12 Female   1992   0 1992      2   0    2 Female
13 Female   1992   1 1993      2   1    3 Female
14 Female   1992   2 1994      2   2    4 Female
15 Female   1992   3 1995      2   3    5 Female
16 Female   1993   0 1993      3   0    3 Female
17 Female   1993   1 1994      3   1    4 Female
18 Female   1993   2 1995      3   2    5 Female
19 Female   1994   0 1994      4   0    4 Female
20 Female   1994   1 1995      4   1    5 Female
21 Female   1995   0 1995      5   0    5 Female
22   Male   1990   0 1990      0   0    0   Male
23   Male   1990   1 1991      0   1    1   Male
24   Male   1990   2 1992      0   2    2   Male
25   Male   1990   3 1993      0   3    3   Male
26   Male   1990   4 1994      0   4    4   Male
27   Male   1990   5 1995      0   5    5   Male
28   Male   1991   0 1991      1   0    1   Male
29   Male   1991   1 1992      1   1    2   Male
30   Male   1991   2 1993      1   2    3   Male
31   Male   1991   3 1994      1   3    4   Male
32   Male   1991   4 1995      1   4    5   Male
33   Male   1992   0 1992      2   0    2   Male
34   Male   1992   1 1993      2   1    3   Male
35   Male   1992   2 1994      2   2    4   Male
36   Male   1992   3 1995      2   3    5   Male
37   Male   1993   0 1993      3   0    3   Male
38   Male   1993   1 1994      3   1    4   Male
39   Male   1993   2 1995      3   2    5   Male
40   Male   1994   0 1994      4   0    4   Male
41   Male   1994   1 1995      4   1    5   Male
42   Male   1995   0 1995      5   0    5   Male
>
```

Thus, row 1 in the Phi design data is for parameter 1 which is the Female survival from 1990 to 1991 of the 1990 cohort and row 29 is for parameter 29 which is survival of males from the 1991 cohort from 1992 to 1993. RMark assumes the design data is in the numeric order of parameters. **If you were to sort the design data in a different order, expect bad things to happen. Do not sort the design data!** I need to add an error check to guard against that because later when we discuss merging other covariates with the design data, the potential exists to inadvertently sort the design data.

Now if we consider the design data for p it is essentially the same except that the rows of the design data are numbered from 1 to 42 rather than using the all-different parameter numbers that range from 43-84. This is the index difference that I mentioned earlier.

```
> PIMS(myresults[[1]],"p",simplified=F)
group = sexFemale
     1991 1992 1993 1994 1995 1996
1990   43   44   45   46   47   48
1991        49   50   51   52   53
1992             54   55   56   57
1993                  58   59   60
1994                       61   62
1995                            63
group = sexMale
     1991 1992 1993 1994 1995 1996
1990   64   65   66   67   68   69
1991        70   71   72   73   74
1992             75   76   77   78
1993                  79   80   81
1994                       82   83
1995                            84
```

```
> newdipper.ddl$p
    group cohort age time Cohort Age Time    sex
1  Female   1990   1 1991      0   1    0 Female
2  Female   1990   2 1992      0   2    1 Female
3  Female   1990   3 1993      0   3    2 Female
4  Female   1990   4 1994      0   4    3 Female
5  Female   1990   5 1995      0   5    4 Female
6  Female   1990   6 1996      0   6    5 Female
7  Female   1991   1 1992      1   1    1 Female
8  Female   1991   2 1993      1   2    2 Female
9  Female   1991   3 1994      1   3    3 Female
10 Female   1991   4 1995      1   4    4 Female
11 Female   1991   5 1996      1   5    5 Female
12 Female   1992   1 1993      2   1    2 Female
13 Female   1992   2 1994      2   2    3 Female
14 Female   1992   3 1995      2   3    4 Female
15 Female   1992   4 1996      2   4    5 Female
16 Female   1993   1 1994      3   1    3 Female
17 Female   1993   2 1995      3   2    4 Female
18 Female   1993   3 1996      3   3    5 Female
19 Female   1994   1 1995      4   1    4 Female
20 Female   1994   2 1996      4   2    5 Female
21 Female   1995   1 1996      5   1    5 Female
22   Male   1990   1 1991      0   1    0   Male
23   Male   1990   2 1992      0   2    1   Male
24   Male   1990   3 1993      0   3    2   Male
25   Male   1990   4 1994      0   4    3   Male
26   Male   1990   5 1995      0   5    4   Male
27   Male   1990   6 1996      0   6    5   Male
28   Male   1991   1 1992      1   1    1   Male
29   Male   1991   2 1993      1   2    2   Male
30   Male   1991   3 1994      1   3    3   Male
31   Male   1991   4 1995      1   4    4   Male
32   Male   1991   5 1996      1   5    5   Male
33   Male   1992   1 1993      2   1    2   Male
34   Male   1992   2 1994      2   2    3   Male
35   Male   1992   3 1995      2   3    4   Male
36   Male   1992   4 1996      2   4    5   Male
37   Male   1993   1 1994      3   1    3   Male
38   Male   1993   2 1995      3   2    4   Male
39   Male   1993   3 1996      3   3    5   Male
40   Male   1994   1 1995      4   1    4   Male
41   Male   1994   2 1996      4   2    5   Male
42   Male   1995   1 1996      5   1    5   Male
>
```

So rows 1 and 29 of the design data refer to the same position but the unique parameter number is the row number plus 42 because the p parameters are numbered after the Phi parameters. Notice that time and age are labeled differently for Phi and p. This is a convention I chose to label interval parameters like Phi (survival in the interval between 2 occasions) with the time or age of the occasion that begins the interval. Whereas, p is a parameter for a specific occasion, so it is labeled by the time/age at that occasion. Because it is just labeling you can modify the design data if you so choose.

In contemplating how to write this section, I stumbled on a use for the `tapply` function that will hopefully help further clarify the link between PIMs and design data. Using that function we can show the design data in the same format as the PIMs. We used the `tapply` function earlier to get the mean weights for covariate predictions. It is a general R function that applies any function to a vector (its first argument) that is split into categories defined in a list with one or more factor variables. For this `tapply` application we will use the `unique` function which returns the unique values in a vector. Since all the values will be the same in the vectors that we give it, it will return a single value. Below I show the design data for time, cohort and age for Phi. I deleted the results for males because they look the same:

```
> with(newdipper.ddl$Phi, tapply(time,list(cohort,time,group),unique))
, , Female

     1990 1991 1992 1993 1994 1995
1990    1    2    3    4    5    6
1991   NA    2    3    4    5    6
1992   NA   NA    3    4    5    6
1993   NA   NA   NA    4    5    6
1994   NA   NA   NA   NA    5    6
1995   NA   NA   NA   NA   NA    6
…

> with(newdipper.ddl$Phi, tapply(cohort,list(cohort,time,group),unique))
, , Female

     1990 1991 1992 1993 1994 1995
1990    1    1    1    1    1    1
1991   NA    2    2    2    2    2
1992   NA   NA    3    3    3    3
1993   NA   NA   NA    4    4    4
1994   NA   NA   NA   NA    5    5
1995   NA   NA   NA   NA   NA    6

…

> with(newdipper.ddl$Phi, tapply(age,list(cohort,time,group),unique))
, , Female

     1990 1991 1992 1993 1994 1995
1990    1    2    3    4    5    6
1991   NA    1    2    3    4    5
1992   NA   NA    1    2    3    4
1993   NA   NA   NA    1    2    3
1994   NA   NA   NA   NA    1    2
1995   NA   NA   NA   NA   NA    1
…
```

Notice that time is the column number, cohort is the row number and age is the same across diagonals. What is shown is the numeric value of the factor variable. If instead we wanted to see the levels (labels) for the factor variables then we could do the following which also shows them for group as well:

```
> with(newdipper.ddl$Phi, tapply(levels(group)[group],list(cohort,time,group),unique))
, , Female

     1990     1991     1992     1993     1994     1995
1990 "Female" "Female" "Female" "Female" "Female" "Female"
1991 NA       "Female" "Female" "Female" "Female" "Female"
1992 NA       NA       "Female" "Female" "Female" "Female"
1993 NA       NA       NA       "Female" "Female" "Female"
1994 NA       NA       NA       NA       "Female" "Female"
1995 NA       NA       NA       NA       NA       "Female"

, , Male

     1990   1991   1992   1993   1994   1995
1990 "Male" "Male" "Male" "Male" "Male" "Male"
1991 NA     "Male" "Male" "Male" "Male" "Male"
1992 NA     NA     "Male" "Male" "Male" "Male"
1993 NA     NA     NA     "Male" "Male" "Male"
1994 NA     NA     NA     NA     "Male" "Male"
1995 NA     NA     NA     NA     NA     "Male"

> with(newdipper.ddl$Phi, tapply(levels(cohort)[cohort],list(cohort,time,group),unique))
, , Female

     1990   1991   1992   1993   1994   1995
1990 "1990" "1990" "1990" "1990" "1990" "1990"
1991 NA     "1991" "1991" "1991" "1991" "1991"
1992 NA     NA     "1992" "1992" "1992" "1992"
1993 NA     NA     NA     "1993" "1993" "1993"
1994 NA     NA     NA     NA     "1994" "1994"
1995 NA     NA     NA     NA     NA     "1995"

, , Male
```

```
     1990   1991   1992   1993   1994   1995
1990 "1990" "1990" "1990" "1990" "1990" "1990"
1991 NA     "1991" "1991" "1991" "1991" "1991"
1992 NA     NA     "1992" "1992" "1992" "1992"
1993 NA     NA     NA     "1993" "1993" "1993"
1994 NA     NA     NA     NA     "1994" "1994"
1995 NA     NA     NA     NA     NA     "1995"

> with(newdipper.ddl$Phi, tapply(levels(time)[time],list(cohort,time,group),unique))
, , Female

     1990   1991   1992   1993   1994   1995
1990 "1990" "1991" "1992" "1993" "1994" "1995"
1991 NA     "1991" "1992" "1993" "1994" "1995"
1992 NA     NA     "1992" "1993" "1994" "1995"
1993 NA     NA     NA     "1993" "1994" "1995"
1994 NA     NA     NA     NA     "1994" "1995"
1995 NA     NA     NA     NA     NA     "1995"

, , Male

     1990   1991   1992   1993   1994   1995
1990 "1990" "1991" "1992" "1993" "1994" "1995"
1991 NA     "1991" "1992" "1993" "1994" "1995"
1992 NA     NA     "1992" "1993" "1994" "1995"
1993 NA     NA     NA     "1993" "1994" "1995"
1994 NA     NA     NA     NA     "1994" "1995"
1995 NA     NA     NA     NA     NA     "1995"

> with(newdipper.ddl$Phi, tapply(levels(age)[age],list(cohort,time,group),unique))
, , Female

     1990 1991 1992 1993 1994 1995
1990 "0"  "1"  "2"  "3"  "4"  "5"
1991 NA   "0"  "1"  "2"  "3"  "4"
1992 NA   NA   "0"  "1"  "2"  "3"
1993 NA   NA   NA   "0"  "1"  "2"
1994 NA   NA   NA   NA   "0"  "1"
1995 NA   NA   NA   NA   NA   "0"

, , Male

     1990 1991 1992 1993 1994 1995
1990 "0"  "1"  "2"  "3"  "4"  "5"
1991 NA   "0"  "1"  "2"  "3"  "4"
1992 NA   NA   "0"  "1"  "2"  "3"
1993 NA   NA   NA   "0"  "1"  "2"
1994 NA   NA   NA   NA   "0"  "1"
1995 NA   NA   NA   NA   NA   "0"
```

Notice that the age variable looks the same because the initial age at release is assumed to be 0. We'll discuss age and age models a little later on.

c. Adding design data

i. Using add.design.data

You are not limited to the default set of design data that are created. You can add as many fields (columns) as you would like. The function `add.design.data` was created to provide a simple way of adding factor variables that binned time, cohort or age into contiguous intervals that the user specifies. It has rather limited usefulness and its name suggests that it does more than it actually can. An example for the `newdipper` data might be to create an `ageclass` variable for survival that was juvenile (0) and adult (1+):

```
> newdipper.ddl=add.design.data(newdipper.proc,newdipper.ddl,"Phi",
                 type="age",bins=c(0,1,8),name="ageclass",right=FALSE)
```

The `bins` argument specifies the cut points and the default is for the intervals to be closed on the right (i.e., includes right end point in the interval) and open on

the left (doesn't include the left end point in the interval) indicated by (…]. If you specify `right=FALSE`, then the intervals are closed on the left and open on the right indicated by […). The first interval is always closed on the left and the last interval is always closed on the right. So with this example and right=FALSE, the intervals are [0,1) and [1,8] which means the first interval includes 0 and the second includes all other values (1+). We can see that by summarizing the Phi design data:

```
> summary(newdipper.ddl$Phi)
    group     cohort   age      time        Cohort          Age
 Female:21   1990:12   0:12   1990: 2   Min.   :0.000   Min.   :0.000
 Male  :21   1991:10   1:10   1991: 4   1st Qu.:0.000   1st Qu.:0.000
             1992: 8   2: 8   1992: 6   Median :1.000   Median :1.000
             1993: 6   3: 6   1993: 8   Mean   :1.667   Mean   :1.667
             1994: 4   4: 4   1994:10   3rd Qu.:3.000   3rd Qu.:3.000
             1995: 2   5: 2   1995:12   Max.   :5.000   Max.   :5.000
     Time           sex      ageclass
 Min.   :0.000   Female:21   [0,1):12
 1st Qu.:2.000   Male  :21   [1,8]:30
 Median :4.000
 Mean   :3.333
 3rd Qu.:5.000
 Max.   :5.000

> levels(newdipper.ddl$Phi$ageclass)=c("juvenile","adult")
> summary(newdipper.ddl$Phi)
    group     cohort   age      time        Cohort          Age
 Female:21   1990:12   0:12   1990: 2   Min.   :0.000   Min.   :0.000
 Male  :21   1991:10   1:10   1991: 4   1st Qu.:0.000   1st Qu.:0.000
             1992: 8   2: 8   1992: 6   Median :1.000   Median :1.000
             1993: 6   3: 6   1993: 8   Mean   :1.667   Mean   :1.667
             1994: 4   4: 4   1994:10   3rd Qu.:3.000   3rd Qu.:3.000
             1995: 2   5: 2   1995:12   Max.   :5.000   Max.   :5.000
     Time           sex         ageclass
 Min.   :0.000   Female:21   juvenile:12
 1st Qu.:2.000   Male  :21   adult   :30
 Median :4.000
 Mean   :3.333
 3rd Qu.:5.000
 Max.   :5.000
```

Now the variable `ageclass` with only 2 levels could be used in place of `age` with 6 levels. Binning is useful to reduce the number of estimated parameters when you expect no difference in the real parameter across adjoining factors levels. For example, we might expect survival of juveniles might be different than adults but no difference amongst adults of different ages.

ii. Using R commands

In cases where you want to combine factor levels that are not adjoining like above then you can use the R language/functions to create the design data however you want. For example, what if we thought that dipper survival might depend on the occurrence of stream flooding and I knew that stream flooding occurred in 1992 and 1993 but not in the other years. Then I could use the following approach to define a design data field I call `flood`:

```
> newdipper.ddl$Phi$flood=0
> newdipper.ddl$Phi$flood[newdipper.ddl$Phi$time==1992 |
                          newdipper.ddl$Phi$time==1993]=1
> head(newdipper.ddl$Phi,10)
```

```
    group cohort age time Cohort Age Time    sex ageclass flood
1  Female   1990   0 1990      0   0    0 Female juvenile     0
2  Female   1990   1 1991      0   1    1 Female    adult     0
3  Female   1990   2 1992      0   2    2 Female    adult     1
4  Female   1990   3 1993      0   3    3 Female    adult     1
5  Female   1990   4 1994      0   4    4 Female    adult     0
6  Female   1990   5 1995      0   5    5 Female    adult     0
7  Female   1991   0 1991      1   0    1 Female juvenile     0
8  Female   1991   1 1992      1   1    2 Female    adult     1
9  Female   1991   2 1993      1   2    3 Female    adult     1
10 Female   1991   3 1994      1   3    4 Female    adult     0
```

The first line created a variable in the Phi design data named `flood` and assigned the value 0 to each record. The second line took the subset of records which had a value for `time` that was either 1992 or 1993 and assigned the value 1 to `flood` for those records. Note that because time is a factor variable the following would not work because you cannot use relational operators (<, ≤,>, ≥) on factors:

```
> newdipper.ddl$Phi$flood[newdipper.ddl$Phi$time>1992 &
newdipper.ddl$Phi$time<1993]=1
Warning messages:
1: In Ops.factor(newdipper.ddl$Phi$time, 1992) :
  > not meaningful for factors
2: In Ops.factor(newdipper.ddl$Phi$time, 1993) :
  < not meaningful for factors
```

If you want to use a relational operator on `time`, `age` or `cohort`, use their numerical equivalents `Time`, `Age` and `Cohort`. If you have used MARK, you may recognize that the `flood` column looks like a column from a design matrix which we have not yet discussed. In fact, because it is a 0/1 numeric variable it will effectively be treated as a pre-defined column in the design matrix. This type of 0/1 dummy variable can be quite useful to construct complicated design matrices with formulas.

iii. Using merge.design.covariates

A situation may arise where you have many different covariates that you want to append to the design data and it is easier to put them in a dataframe and merge them with the design data. For this task, you can use the function called `merge.design.covariates`. As an example, I'll create some fake effort data as a possible explanatory variable for capture probability. In addition, I'll add a weather factor variable with values sunny, cloudy and rainy. These variables vary by time (occasion) but are the same for each `group`(`sex`). So I can create the following dataframe:

```
> mypvars=data.frame(time=1991:1996,effort=c(12,4,22,11,65,2),
weather=c("cloudy","rainy","sunny","cloudy","rainy","sunny"))
> mypvars
  time effort weather
1 1991     12  cloudy
2 1992      4   rainy
3 1993     22   sunny
4 1994     11  cloudy
5 1995     65   rainy
6 1996      2   sunny
```

The actual values here are not important but I chose a numeric variable (`effort`) and a factor variable (`weather`), to show that either can be used and they will remain in the design data the way that you define them in the dataframe. To keep it simple, I created the dataframe with code, but you could also read it in from a file as well. One of the important features of this dataframe is that it has a `time` field and their values will match the values of the `time` field in the design data for p. If the design data fields that I was adding also varied by group, then I would also have to have a `group` field in the dataframe that matched the values of the `group` field in the design data. For the current example I can add the new fields effort and weather to my design data with the following command:

```
> newdipper.ddl$p=merge.design.covariates(newdipper.ddl$p,mypvars,
                          bygroup=FALSE,bytime=TRUE)
> newdipper.ddl$p
   time  group cohort age Cohort Age Time    sex effort weather
1  1991 Female   1990   1      0   1    0 Female     12  cloudy
2  1992 Female   1990   2      0   2    1 Female      4   rainy
3  1993 Female   1990   3      0   3    2 Female     22   sunny
4  1994 Female   1990   4      0   4    3 Female     11  cloudy
5  1995 Female   1990   5      0   5    4 Female     65   rainy
6  1996 Female   1990   6      0   6    5 Female      2   sunny
7  1992 Female   1991   1      1   1    1 Female      4   rainy
8  1993 Female   1991   2      1   2    2 Female     22   sunny
9  1994 Female   1991   3      1   3    3 Female     11  cloudy
10 1995 Female   1991   4      1   4    4 Female     65   rainy
11 1996 Female   1991   5      1   5    5 Female      2   sunny
12 1993 Female   1992   1      2   1    2 Female     22   sunny
13 1994 Female   1992   2      2   2    3 Female     11  cloudy
14 1995 Female   1992   3      2   3    4 Female     65   rainy
15 1996 Female   1992   4      2   4    5 Female      2   sunny
16 1994 Female   1993   1      3   1    3 Female     11  cloudy
17 1995 Female   1993   2      3   2    4 Female     65   rainy
18 1996 Female   1993   3      3   3    5 Female      2   sunny
19 1995 Female   1994   1      4   1    4 Female     65   rainy
20 1996 Female   1994   2      4   2    5 Female      2   sunny
21 1996 Female   1995   1      5   1    5 Female      2   sunny
22 1991   Male   1990   1      0   1    0   Male     12  cloudy
23 1992   Male   1990   2      0   2    1   Male      4   rainy
24 1993   Male   1990   3      0   3    2   Male     22   sunny
25 1994   Male   1990   4      0   4    3   Male     11  cloudy
…
```

Occasionally, you may need to add occasion data in this fashion and it may not fit the mold where it varies, either by group, or by time or by group and time. For example, the covariates might depend on `cohort`, `group` and `time`. A word of warning is needed here. The `merge` function in R, will not maintain the original sort order. You can maintain the order by adding a sequential number to the design data, do the `merge` and then return to the original order with the sequence number. You'll also have to keep and restore the original `row.names` if you have deleted design data. For example, assume I have a dataframe called covariates with a field named `fac` and I composed a similar field called `fac` in the Phi design data that pasted together `group`, `cohort` and `time`. Then I could use the following code to merge `covariates` into the Phi design data and maintain the sort order and the `row.names`:

```
save.row.names = row.names(my.ddl$Phi)
my.ddl$Phi$sequence = 1:dim(my.ddl$Phi)[1]
my.ddl$Phi=merge(my.ddl$Phi, covariates,by.x="fac",by.y="fac")
my.ddl$Phi = my.ddl$Phi[order(my.ddl$Phi$sequence), ]
my.ddl$Phi$sequence = NULL
row.names(my.ddl$Phi)=save.row.names
```

d. Age models

So far I have only mentioned the issue of age but haven't talked about how you might use it to create models with age-specific estimate of survival or capture probability. But before we go there we need to describe what we mean by age. There are at least 2 possible meanings. The first and most obvious is the age of an animal or amount of time that elapsed since its birth. The second and possibly less obvious is the age or time since the animal was first captured or first released. If you look at the Cooch and White electronic book, they call this latter age, time since marking or TSM. It has also been called time at liberty meaning the amount of time that elapsed since it was released. Obviously these quantities are related in that an animal's absolute age at any time during the experiment is its initial age at capture (release) plus TSM. If all of the animals are the same age (from birth) at time of release (or initial capture), then absolute age and TSM are only separated by a constant and they are effectively the same quantity. However, if the initial ages of animals at release are different, then absolute age and TSM are different quantities in that I can have 2 animals with the same TSM and different absolute ages and vice versa.

If you want to use absolute age in models and the animals have different initial ages at release (initial capture), then you need to separate the animals into groups with a factor variable such that each group is composed of a single initial age. Then each group is assigned an initial age in `process.data`. For example, I'll use `newdipper` and create yet some more fake data to assign dippers into hatch-year and adult initial age classes and use that to create groups and design data. I created the variable `class` and alternated in the data between Adult (1+) and Hatch-year (0).

```
> newdipper$class=factor(rep(c("Adult","Hatch-year"),each=294/2))

> newdipper.proc=process.data(newdipper,begin.time=1990,groups=c("sex","class"),
                                      age.var=2,initial.age=c(1,0))
```

In the call to `process.data`, I used both `sex` and `class` as factor variables for `groups`. Because my variable for `age` is not named age (I did that on purpose), I had to tell it that the second factor variable should be used to assign initial ages (`age.var=2`). Then I assigned initial ages for the 2 levels of `class` by `initial.age=c(1,0)`. The initial age values are assigned in the order of the levels of `class` which is "Adult", "Hatch-year" because the default order is alphabetical. Notice that even though I have 4 groups in the data (2 `sex` * 2 `class`), I only have to assign 2 values for `initial.age` for the 2 levels of `class`.

I could have used the `levels` argument of `factor` to change the order of the levels of `class` in `newdipper` prior to calling `process.data`. Now when I create the design data, you can see that those with `class="Adult"` start at an initial age of 1 in their first year and their age subsequently increases in the subsequent years and those with `class="Hatch-year"` start at an initial age of 0 in their first year.

```
> newdipper.ddl=make.design.data(newdipper.proc)
> newdipper.ddl$Phi
              group cohort age time Cohort Age Time    sex      class
1       FemaleAdult   1990   1 1990      0   1    0 Female      Adult
2       FemaleAdult   1990   2 1991      0   2    1 Female      Adult
3       FemaleAdult   1990   3 1992      0   3    2 Female      Adult
4       FemaleAdult   1990   4 1993      0   4    3 Female      Adult
5       FemaleAdult   1990   5 1994      0   5    4 Female      Adult
6       FemaleAdult   1990   6 1995      0   6    5 Female      Adult
7       FemaleAdult   1991   1 1991      1   1    1 Female      Adult
8       FemaleAdult   1991   2 1992      1   2    2 Female      Adult
9       FemaleAdult   1991   3 1993      1   3    3 Female      Adult
10      FemaleAdult   1991   4 1994      1   4    4 Female      Adult
…
43 FemaleHatch-year   1990   0 1990      0   0    0 Female Hatch-year
44 FemaleHatch-year   1990   1 1991      0   1    1 Female Hatch-year
45 FemaleHatch-year   1990   2 1992      0   2    2 Female Hatch-year
46 FemaleHatch-year   1990   3 1993      0   3    3 Female Hatch-year
47 FemaleHatch-year   1990   4 1994      0   4    4 Female Hatch-year
48 FemaleHatch-year   1990   5 1995      0   5    5 Female Hatch-year
49 FemaleHatch-year   1991   0 1991      1   0    1 Female Hatch-year
50 FemaleHatch-year   1991   1 1992      1   1    2 Female Hatch-year
51 FemaleHatch-year   1991   2 1993      1   2    3 Female Hatch-year
52 FemaleHatch-year   1991   3 1994      1   3    4 Female Hatch-year
…
```

Now because adults are really anything that are 1 or older then I don't really know their true initial age beyond being 1+. Thus, it would not make sense to treat `age` as truly being known beyond 0 and 1+. So, this is an example where you would use an option in `make.design.data` to bin the `age` variable (not the `initial.age`) as follows:

```
>newdipper.ddl=make.design.data(newdipper.proc,
                    parameters=list(Phi=list(age.bins=c(0,1,6))),right=FALSE)
> levels(newdipper.ddl$Phi$age)=c("0","1+")
> newdipper.ddl$Phi
               group cohort age time Cohort Age Time    sex      class
1       FemaleAdult   1990  1+ 1990      0   1    0 Female      Adult
2       FemaleAdult   1990  1+ 1991      0   2    1 Female      Adult
3       FemaleAdult   1990  1+ 1992      0   3    2 Female      Adult
4       FemaleAdult   1990  1+ 1993      0   4    3 Female      Adult
5       FemaleAdult   1990  1+ 1994      0   5    4 Female      Adult
6       FemaleAdult   1990  1+ 1995      0   6    5 Female      Adult
…
43 FemaleHatch-year   1990   0 1990      0   0    0 Female Hatch-year
44 FemaleHatch-year   1990  1+ 1991      0   1    1 Female Hatch-year
45 FemaleHatch-year   1990  1+ 1992      0   2    2 Female Hatch-year
46 FemaleHatch-year   1990  1+ 1993      0   3    3 Female Hatch-year
47 FemaleHatch-year   1990  1+ 1994      0   4    4 Female Hatch-year
48 FemaleHatch-year   1990  1+ 1995      0   5    5 Female Hatch-year
49 FemaleHatch-year   1991   0 1991      1   0    1 Female Hatch-year
50 FemaleHatch-year   1991  1+ 1992      1   1    2 Female Hatch-year
```

Notice that the `age` variable transitions in the design data from 0 to 1+ for Hatch-year birds but the original `class` variable is static. Don't get confused between `age` and `initial.age`. The former is dynamic and the latter is static.

IX. Digging into the details of RMark: Formula and design matrices

a. *What is a design matrix?*

A design matrix and a vector of parameters provide a compact form for a series of equations. Each row in the design matrix is for one of the equations. The number of columns in a design matrix matches the number of values in the parameter vector. Each equation is created by multiplying the respective elements of each design matrix row and the parameter vector and adding them up. That is the definition of matrix multiplication. For example, imagine that we had 5 values for y (y_1, y_2, y_3, y_4, y_5) and x (x_1, x_2, x_3, x_4, x_5) and we wanted to explain the values of y by a line using x: y=a+b*x. We have 5 equations so we have 5 rows in our design matrix and we have 2 parameters (a,b) so we'll have 2 columns in the design matrix. We can represent the 5 equations as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ 1 & x_4 \\ 1 & x_5 \end{bmatrix} \times \begin{bmatrix} a & b \end{bmatrix}$$

So for example, the equation for row 3 is y_3 =a+b*x_3. For MARK, (a,b) are equivalent to the beta parameters and if an identity link was used the y's are equivalent to the real parameters. An identity link means there is no transformation which forces constraints on parameters. In general a non-identity link is used to constrain parameters to be probabilities (e.g., logit link to constrain real parameter between 0 and 1) or strictly positive (e.g., log link to constrain abundance to be greater than 0). With a non-identity link, a+b*x is the value of the link equation and not the real parameter value. For example, if we used the logit link function then the equation for y_3 would be the inverse of the logit link:

$$y_3 = \frac{1}{1 + e^{-a-b*x_3}}$$

and the logit link applied to y_3 yields the linear equation represented by the betas and the design matrix:

$$a + b * x_3 = \log\left(\frac{y_3}{1 - y_3}\right)$$

Other link functions have different equations, but the idea is the same. Each element of the beta parameter vector is multiplied by the respective values in the row of the design matrix and then this goes into the inverse-link transformation to yield the real parameter. This is done for every row of the design matrix to yield each of the real parameter values.

If you have used MARK very much then you probably knew all of that and you also know that MARK requires you to create the design matrix by filling in each of the values of the matrix. RMark avoids that by using a formula with the design data to create the design matrix.

b. How is the design matrix created from the formula?

A magical little R function called `model.matrix` makes all of this possible. It takes a formula and a dataframe and creates the design matrix. For RMark, the dataframe is one of the design dataframes in the design data list (ddl) for a parameter like Phi or p. The formula is something that you create to describe the model that you want to describe your data using the variables in the design data or the individual covariates in your data. To help you understand the links between formula, design data and design matrix, it is useful to use `model.matrix` in R to see what design matrices it creates with various formulas. In particular it is important to understand the different ways that numeric and factor variables are treated and how to specify and understand interactions.

So to start, let's create a design matrix for p which has a linear trend over time using the numeric Time variable. To save paper (or disk space), I'm going to use the `head` function to limit the list to the first 10 entries in the matrix:

```
> head(model.matrix(~Time,newdipper.ddl$p),10)
   (Intercept) Time
1            1    0
2            1    1
3            1    2
4            1    3
5            1    4
6            1    5
7            1    1
8            1    2
9            1    3
10           1    4
```

The first 10 rows of the design data which correspond to these 10 rows of the design matrix are:

```
> head(newdipper.ddl$p,10)
         group cohort age time Cohort Age Time    sex class
1  FemaleAdult   1990   2 1991      0   2    0 Female Adult
2  FemaleAdult   1990   3 1992      0   3    1 Female Adult
3  FemaleAdult   1990   4 1993      0   4    2 Female Adult
4  FemaleAdult   1990   5 1994      0   5    3 Female Adult
5  FemaleAdult   1990   6 1995      0   6    4 Female Adult
6  FemaleAdult   1990   7 1996      0   7    5 Female Adult
7  FemaleAdult   1991   2 1992      1   2    1 Female Adult
8  FemaleAdult   1991   3 1993      1   3    2 Female Adult
9  FemaleAdult   1991   4 1994      1   4    3 Female Adult
10 FemaleAdult   1991   5 1995      1   5    4 Female Adult
```

Our link equation for this model is simply, a+b*Time or if we use the proper symbol for beta (β), use a subscript index that corresponds to the column number and drop the * and implicitly assume multiplication, the equation is $\beta_1 + \beta_2 Time$. If we were to assume that the default logit link was used then the equation for each p would be:

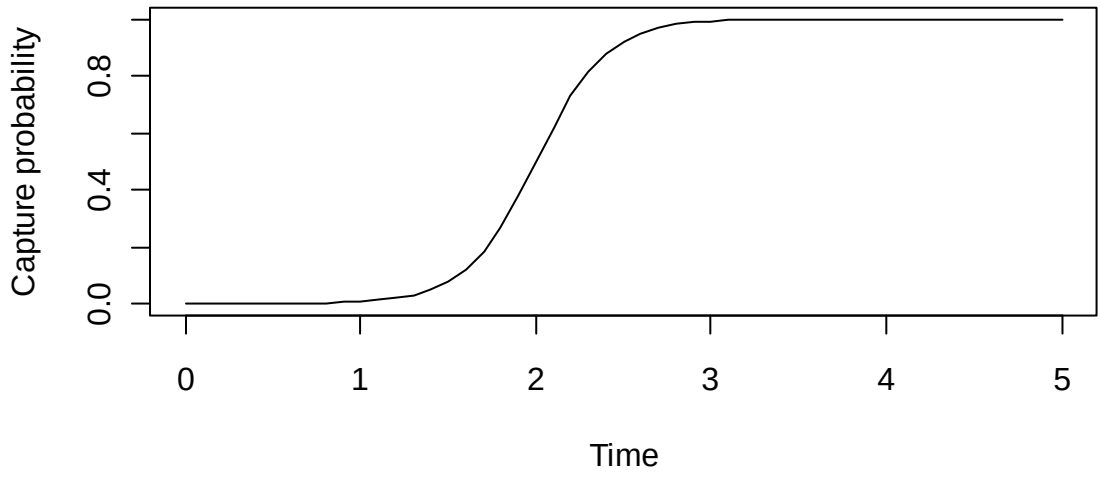
$$p_i = \frac{1}{1 + e^{-\beta_1 - \beta_2 Time_i}}$$

where the index *i* is the row number in the design data. So for i=10, the formula is:

$$p_{10} = \frac{1}{1 + e^{-\beta_1 - \beta_2 4}}$$

In this case, β_1 is the intercept which is the value when Time=0 and β_2 is the slope for Time. It is important to recognize that the equation is a line but not after it is transformed to become a real parameter. Thus, if you were to plot the values of p for each value of time, depending on the quantities it may look close to a line or not, but it is NOT a line. In fact, if the betas and the values of time are such that it produces estimates from p=0 to p=1, it will not look anything like a line and it will look like a logistic curve.

```
> plot((0:50)/10,plogis(-10+5*(0:50)/10),type="l",xlab="Time",ylab="Capture probability")
```



Before going any further, I want to tie this back into the PIMs. For this model, if we were to compute each equation they would look as follows for p for any of the age-sex groups in the newdipper data.

1991	1992	1993	1994	1995	1996
$\frac{1}{1+e^{-\beta_1}}$	$\frac{1}{1+e^{-\beta_1-\beta_2}}$	$\frac{1}{1+e^{-\beta_1-2\beta_2}}$	$\frac{1}{1+e^{-\beta_1-3\beta_2}}$	$\frac{1}{1+e^{-\beta_1-4\beta_2}}$	$\frac{1}{1+e^{-\beta_1-5\beta_2}}$
	$\frac{1}{1+e^{-\beta_1-\beta_2}}$	$\frac{1}{1+e^{-\beta_1-2\beta_2}}$	$\frac{1}{1+e^{-\beta_1-3\beta_2}}$	$\frac{1}{1+e^{-\beta_1-4\beta_2}}$	$\frac{1}{1+e^{-\beta_1-5\beta_2}}$
		$\frac{1}{1+e^{-\beta_1-2\beta_2}}$	$\frac{1}{1+e^{-\beta_1-3\beta_2}}$	$\frac{1}{1+e^{-\beta_1-4\beta_2}}$	$\frac{1}{1+e^{-\beta_1-5\beta_2}}$
			$\frac{1}{1+e^{-\beta_1-3\beta_2}}$	$\frac{1}{1+e^{-\beta_1-4\beta_2}}$	$\frac{1}{1+e^{-\beta_1-5\beta_2}}$
				$\frac{1}{1+e^{-\beta_1-4\beta_2}}$	$\frac{1}{1+e^{-\beta_1-5\beta_2}}$
					$\frac{1}{1+e^{-\beta_1-5\beta_2}}$

For each column the equation is the same because Time is the same in each column. Each of the rows in the design data are tied to the real parameters via the index (number) and this provides the mechanism to specify how the p vary by time, cohort, age, by any factor variables used for groups, and by any other design data that you create.

Now let's consider a model with the factor variable `time`. In creating a design matrix from a factor variable, it is important to realize that you can construct different design matrices that represent the exact same equation values. The typical way to construct the design matrix is to choose one of the factor levels as the intercept and to represent each of the other parameters for the remaining factor levels as an amount that is added onto the intercept. This is called a treatment contrast and is the default used by `model.matrix`. Even within this convention, there are differences in the choice of the factor level that is chosen as the intercept. In creating its pre-defined models, MARK uses the SAS software convention of using the last level of the factor variable for the intercept, whereas R uses the first level of the factor variable. This does not influence the real parameter estimates but does change your interpretation and value of the beta parameters. With this convention, except for the intercept, the columns of the design matrix become dummy variables (0/1) that represent absence/presence of each factor level. For example, let's consider the `~time` model for p and look at the first 10 rows of the design matrix:

```
> head(model.matrix(~time,newdipper.ddl$p),10)
   (Intercept) time1992 time1993 time1994 time1995 time1996
1            1        0        0        0        0        0
2            1        1        0        0        0        0
3            1        0        1        0        0        0
4            1        0        0        1        0        0
5            1        0        0        0        1        0
6            1        0        0        0        0        1
7            1        1        0        0        0        0
8            1        0        1        0        0        0
9            1        0        0        1        0        0
10           1        0        0        0        1        0
> head(newdipper.ddl$p,10)
         group cohort age time Cohort Age Time    sex class
1  FemaleAdult   1990   2 1991      0   2    0 Female Adult
2  FemaleAdult   1990   3 1992      0   3    1 Female Adult
3  FemaleAdult   1990   4 1993      0   4    2 Female Adult
4  FemaleAdult   1990   5 1994      0   5    3 Female Adult
```

```
5  FemaleAdult   1990   6 1995      0   6    4 Female Adult
6  FemaleAdult   1990   7 1996      0   7    5 Female Adult
7  FemaleAdult   1991   2 1992      1   2    1 Female Adult
8  FemaleAdult   1991   3 1993      1   3    2 Female Adult
9  FemaleAdult   1991   4 1994      1   4    3 Female Adult
10 FemaleAdult   1991   5 1995      1   5    4 Female Adult
```

Instead of 2 parameters with the ~Time model we now have 6 beta parameters because we have 6 recapture occasions and thus 6 levels for time. Like with the ~Time model, the intercept (β_1) is for the first recapture occasion which is in 1991. If you were to multiply the beta vector (β_1, β_2, β_3, β_4, β_5, β_6) times row 10 you'll get $\beta_1 + \beta_5$ which is the link value for the 5th occasion (time=1995). The estimated capture probability for 1995 is:

$$p_{1995} = \frac{1}{1 + e^{-\beta_1 - \beta_5}}$$

That value for p would be used for row 10 and anywhere else that time=1995 which would be the 5th column in a PIM representation. So in PIM format the equations would be:

1991	**1992**	**1993**	**1994**	**1995**	**1996**
$\frac{1}{1+e^{-\beta_1}}$	$\frac{1}{1+e^{-\beta_1-\beta_2}}$	$\frac{1}{1+e^{-\beta_1-\beta_3}}$	$\frac{1}{1+e^{-\beta_1-\beta_4}}$	$\frac{1}{1+e^{-\beta_1-\beta_5}}$	$\frac{1}{1+e^{-\beta_1-\beta_6}}$
	$\frac{1}{1+e^{-\beta_1-\beta_2}}$	$\frac{1}{1+e^{-\beta_1-\beta_3}}$	$\frac{1}{1+e^{-\beta_1-\beta_4}}$	$\frac{1}{1+e^{-\beta_1-\beta_5}}$	$\frac{1}{1+e^{-\beta_1-\beta_6}}$
		$\frac{1}{1+e^{-\beta_1-\beta_3}}$	$\frac{1}{1+e^{-\beta_1-\beta_4}}$	$\frac{1}{1+e^{-\beta_1-\beta_5}}$	$\frac{1}{1+e^{-\beta_1-\beta_6}}$
			$\frac{1}{1+e^{-\beta_1-\beta_4}}$	$\frac{1}{1+e^{-\beta_1-\beta_5}}$	$\frac{1}{1+e^{-\beta_1-\beta_6}}$
				$\frac{1}{1+e^{-\beta_1-\beta_5}}$	$\frac{1}{1+e^{-\beta_1-\beta_6}}$
					$\frac{1}{1+e^{-\beta_1-\beta_6}}$

For factor variables, the columns of the design matrix select with 0s and 1s the parameters to combine for that factor.

What happens when I use ~-1+time? How does it change the design matrix? Remember I said that -1 removes the intercept. Let's look:

```
> head(model.matrix(~-1+time,newdipper.ddl$p),10)
   time1991 time1992 time1993 time1994 time1995 time1996
1         1        0        0        0        0        0
2         0        1        0        0        0        0
3         0        0        1        0        0        0
4         0        0        0        1        0        0
5         0        0        0        0        1        0
6         0        0        0        0        0        1
7         0        1        0        0        0        0
8         0        0        1        0        0        0
9         0        0        0        1        0        0
10        0        0        0        0        1        0
```

When you include -1 and a factor variable, it will create separate beta parameters for each factor level, rather than specifying them as an intercept plus another beta. This produces an identity design matrix which means that for each real parameter (each row) all columns are zero except for one column that has a 1. That column links a particular beta parameter to a real parameter. It will produce the same values for p as the `~time` model, but the values for beta will change. To use a sin link, the real parameters have to be specified by an identity design matrix.

Now let's expand the formula to include 2 factor variables in an additive model. To do this we'll use the Phi design data with the formula `~time+age` and we'll look at the first 5 rows which are for Adult Females and rows 43-47 which are the same times for Hatch-year Females:

```
> model.matrix(~time+age,newdipper.ddl$Phi)[c(1:5,43:47),]
   (Intercept) time1991 time1992 time1993 time1994 time1995 age1+
1            1        0        0        0        0        0     1
2            1        1        0        0        0        0     1
3            1        0        1        0        0        0     1
4            1        0        0        1        0        0     1
5            1        0        0        0        1        0     1
43           1        0        0        0        0        0     0
44           1        1        0        0        0        0     1
45           1        0        1        0        0        0     1
46           1        0        0        1        0        0     1
47           1        0        0        0        1        0     1
```

There are several things you need to understand with this example. The first is that when you create an additive model with k factor variables, the number of beta parameters you'll get is the total number of factor levels – k + 1. So for this example, k=2 and the total number of factor levels is 8 (6 for time and 2 for age) and the total number of beta parameters is 7. It helps to view this as a table with the rows being one factor and the columns being the second factor:

time	Age=0	Age=1+
1990	Intercept (β_1)	$\beta_1+\beta_7$
1991	$\beta_1+\beta_2$	$\beta_1+\beta_2+\beta_7$
1992	$\beta_1+\beta_3$	$\beta_1+\beta_3+\beta_7$
1993	$\beta_1+\beta_4$	$\beta_1+\beta_4+\beta_7$
1994	$\beta_1+\beta_5$	$\beta_1+\beta_5+\beta_7$
1995	$\beta_1+\beta_6$	$\beta_1+\beta_6+\beta_7$

The intercept now represents 1990 for age=0 which are both the first levels for the respective factors. The model is additive which means the pattern across times is the same for both age classes and they are only differ by a constant value (β_7). Again this is only true for link values and when these are transformed with the inverse-link function to real parameters the difference will not be constant. One thing else to notice is that row 43 has `age1+` being 0 whereas row 44 also for Hatch-year birds has `age1+` being 1. In 1990 (row 43), the birds were 0 when they were released so their first survival is for `age` 0, but in the following year 1991 (row 44), they were now 1 year old, so they are in the 1+ category. Their initial `class` variable is static but their `age` changes with each year.

Now what if we thought that the pattern across time was not the same for each age? What we are saying is that there is an interaction between age and time. This model can be constructed as ~age*time.

```
> model.matrix(~time*age,newdipper.ddl$Phi)[c(1:6,43:48),]
   (Intercept) time1991 time1992 time1993 time1994 time1995 age1+ time1991:age1+ time1992:age1+ time1993:age1+ time1994:age1+ time1995:age1+
1            1        0        0        0        0        0     1              0              0              0              0              0
2            1        1        0        0        0        0     1              1              0              0              0              0
3            1        0        1        0        0        0     1              0              1              0              0              0
4            1        0        0        1        0        0     1              0              0              1              0              0
5            1        0        0        0        1        0     1              0              0              0              1              0
6            1        0        0        0        0        1     1              0              0              0              0              1
43           1        0        0        0        0        0     0              0              0              0              0              0
44           1        1        0        0        0        0     1              1              0              0              0              0
45           1        0        1        0        0        0     1              0              1              0              0              0
46           1        0        0        1        0        0     1              0              0              1              0              0
47           1        0        0        0        1        0     1              0              0              0              1              0
48           1        0        0        0        0        1     1              0              0              0              0              1
>
```

Now we get 12 beta parameters which is the product of 6*2. You get a beta for every combination of age and time. Now our table will look as follows:

time	Age=0	Age=1+
1990	Intercept (β_1)	$\beta_1+\beta_7$
1991	$\beta_1+\beta_2$	$\beta_1+\beta_2+\beta_7+\beta_8$
1992	$\beta_1+\beta_3$	$\beta_1+\beta_3+\beta_7+\beta_9$
1993	$\beta_1+\beta_4$	$\beta_1+\beta_4+\beta_7+\beta_{10}$
1994	$\beta_1+\beta_5$	$\beta_1+\beta_5+\beta_7+\beta_{11}$
1995	$\beta_1+\beta_6$	$\beta_1+\beta_6+\beta_7+\beta_{12}$

All of the parameters are the same, except that we have now included additional parameters for age 1+ for 1991 to 1995 to allow the time pattern to differ from the time pattern for age 0. The parameters β_2, β_3, β_4, β_5, β_6 are the main-effect parameters for time, β_7 is the main-effect parameter for age, and β_8, β_9, β_{10}, β_{11}, β_{12} are the interaction parameters. The interaction can be viewed as being multiplicative in that the design matrix columns for these last 5 parameters can be obtained by multiplying the columns for the time main-effect (2 to 6) by the age main-effect column (7). The formula ~time*age can be expanded to ~time + age +time:age and the resulting design matrix will be the same:

```
> model.matrix(~time+age+time:age,newdipper.ddl$Phi)[c(1:6,43:48),]
   (Intercept) time1991 time1992 time1993 time1994 time1995 age1+ time1991:age1+ time1992:age1+ time1993:age1+ time1994:age1+ time1995:age1+
1            1        0        0        0        0        0     1              0              0              0              0              0
2            1        1        0        0        0        0     1              1              0              0              0              0
3            1        0        1        0        0        0     1              0              1              0              0              0
4            1        0        0        1        0        0     1              0              0              1              0              0
5            1        0        0        0        1        0     1              0              0              0              1              0
6            1        0        0        0        0        1     1              0              0              0              0              1
43           1        0        0        0        0        0     0              0              0              0              0              0
44           1        1        0        0        0        0     1              1              0              0              0              0
45           1        0        1        0        0        0     1              0              1              0              0              0
46           1        0        0        1        0        0     1              0              0              1              0              0
47           1        0        0        0        1        0     1              0              0              0              1              0
48           1        0        0        0        0        1     1              0              0              0              0              1
>
```

Note that if you change the order to ~age*time, the model will be the same but the column order changes:

```
> model.matrix(~age*time,newdipper.ddl$Phi)[c(1:6,43:48),]
   (Intercept) age1+ time1991 time1992 time1993 time1994 time1995 age1+:time1991 age1+:time1992 age1+:time1993 age1+:time1994 age1+:time1995
1            1     1        0        0        0        0        0              0              0              0              0              0
2            1     1        1        0        0        0        0              1              0              0              0              0
3            1     1        0        1        0        0        0              0              1              0              0              0
4            1     1        0        0        1        0        0              0              0              1              0              0
5            1     1        0        0        0        1        0              0              0              0              1              0
6            1     1        0        0        0        0        1              0              0              0              0              1
```

```
43        1    0       0       0       0       0       0           0           0           0           0           0
44        1    1       1       0       0       0       0           1           0           0           0           0
45        1    1       0       1       0       0       0           0           1           0           0           0
46        1    1       0       0       1       0       0           0           0           1           0           0
47        1    1       0       0       0       1       0           0           0           0           1           0
48        1    1       0       0       0       0       1           0           0           0           0           1
>
```

Now I'll bend your mind a little here. If I use the formula ~time:age I don't just get the last 6 columns as you might expect. You actually get something more useful. You get a unique parameter for each of the 12 factor combinations. Actually you get 13 columns and to be honest I've not figured out why it does this. Fortunately, you can get just the 12 parameters by changing the formula to ~-1+time:age.

```
> model.matrix(~-1+time:age,newdipper.ddl$Phi)[c(1:6,43:48),]
   time1990:age0 time1991:age0 time1992:age0 time1993:age0 time1994:age0 time1995:age0 time1990:age1+
1              0             0             0             0             0             0             1
2              0             0             0             0             0             0             0
3              0             0             0             0             0             0             0
4              0             0             0             0             0             0             0
5              0             0             0             0             0             0             0
6              0             0             0             0             0             0             0
43             1             0             0             0             0             0             0
44             0             0             0             0             0             0             0
45             0             0             0             0             0             0             0
46             0             0             0             0             0             0             0
47             0             0             0             0             0             0             0
48             0             0             0             0             0             0             0
   time1991:age1+ time1992:age1+ time1993:age1+ time1994:age1+ time1995:age1+
1               0             0             0             0             0
2               1             0             0             0             0
3               0             1             0             0             0
4               0             0             1             0             0
5               0             0             0             1             0
6               0             0             0             0             1
43              0             0             0             0             0
44              1             0             0             0             0
45              0             1             0             0             0
46              0             0             1             0             0
47              0             0             0             1             0
48              0             0             0             0             1
```

Not all of the values in the above are represented because we truncated the table. Now our table looks as follows:

time	Age=0	Age=1+
1990	β_1	β_7
1991	β_2	β_8
1992	β_3	β_9
1993	β_4	β_{10}
1994	β_5	β_{11}
1995	β_6	β_{12}

So why would we use this approach? One reason is that it will produce an identity design matrix so you can use the sin link if you choose. But more importantly it will properly handle the case where you don't have data for each cell in your factor table. Imagine what would happen if we only released Hatch-year birds each year. Then in 1990, we would have hatch-year birds but no adults. By using ~-1+time:age, it will only create parameters for the combinations that exist in the design data.

Now let's make this a little more complicated and say that we want ~sex+time*age and assume that we only released Hatch-year birds so what we need to specify is ~sex+time:age. Remember that I said if you use ~-1+sex that it doesn't remove the intercept and only redefines the 2 columns? Well that messes up what we want to do

here which is to have a separate parameter for each time:age combination (excluding age 1+ in 1990) and have an additive sex effect which should only add 1 parameter. Instead either ~sex+time:age or ~-1+sex+time:age will add 2 parameters for sex and the model will have an extra parameter. One solution is to use the `remove.intercept=TRUE` argument for the parameter specification. The definition for Phi would look like:

```
Phi.sex.age.time=list(formula=~sex+time:age, remove.intercept=T)
```

Another approach is to define a numeric variable that I'll call male and define it to be 0 when `sex="Female"` and 1 when `sex="Male"`.

```
> newdipper.ddl$Phi$male=0
> newdipper.ddl$Phi$male[newdipper.ddl$Phi$sex=="Male"]=1
```

Then, you could use `~-1+male+time:age` and you'll get the proper number of parameters.

This brings us back to the use of numeric variables like `male` above which are treated differently than factor variables. As we saw with `~Time` earlier and `~male`, a numeric variable in a formula simply creates a column that contains each value of the numeric variable. There is no creation of dummy variables like with a factor variable. The use of male above illustrates how factor variables can be translated into a set of `m-1` numeric variables where `m` is the number of factor levels. This works best when `m=2` like with `sex` as that only creates a single numeric variable. Another difference with numeric variables is in their use with interactions which are truly multiplicative. If a numeric variable is included in an interaction with another numeric variable the column is simply the product of the two numeric values for each row. An interaction of a numeric and a factor variable is also a product but it is the product of the numeric variable and dummy variables created for the factor variable. For example, let's use a formula `~sex*Time` for p:

```
> model.matrix(~sex*Time,newdipper.ddl$p)[c(1:5,23:27),]
   (Intercept) sexMale Time sexMale:Time
1            1       0    0            0
2            1       0    1            0
3            1       0    2            0
4            1       0    3            0
5            1       0    4            0
23           1       1    1            1
24           1       1    2            2
25           1       1    3            3
26           1       1    4            4
27           1       1    5            5
```

You can think about this model like 2 separate lines with different intercepts (first 2 columns) and different slopes (columns 3 and 4). The line for females is $\beta_1+\beta_3$Time and for males the line $\beta_1+\beta_2+(\beta_3+\beta_4)$Time. This probably doesn't make sense in this case but what if I thought that the slopes were different but the intercepts were the same. I can get that model with `~sex:Time`:

```
> model.matrix(~sex:Time,newdipper.ddl$p)[c(1:5,23:27),]
   (Intercept) sexFemale:Time sexMale:Time
1            1              0            0
2            1              1            0
```

```
3           1             2           0
4           1             3           0
5           1             4           0
23          1             0           1
24          1             0           2
25          1             0           3
26          1             0           4
27          1             0           5
```

Now the line for females is $\beta_1+\beta_2$Time and for males the line $\beta_1+(\beta_2+\beta_3)$Time. The other possibility is that the intercepts are different but the slope is the same and that is the standard additive model ~sex+Time:

```
> model.matrix(~sex+Time,newdipper.ddl$p)[c(1:5,23:27),]
   (Intercept) sexMale Time
1            1       0    0
2            1       0    1
3            1       0    2
4            1       0    3
5            1       0    4
23           1       1    1
24           1       1    2
25           1       1    3
26           1       1    4
27           1       1    5
```

and the line for females is $\beta_1+\beta_3$Time and for males the line is $\beta_1+\beta_2+\beta_3$Time.

Numeric dummy variables can be very useful to construct limited interactions with or among factor variables because the effect will only happen when the numeric variable has a value 1. This can be very useful to limit effects under certain conditions. For example, what if we thought that there was no difference in survival with age but we suspected that due to inexperience the occurrence of stream flooding might impact 0 age birds more than 1+ year birds. We could fit that model as follows:

```
> newdipper.ddl$Phi$flood=0
> newdipper.ddl$Phi$flood[newdipper.ddl$Phi$time==1992 |
                          newdipper.ddl$Phi$time==1993]=1
> model.matrix(~flood:age,newdipper.ddl$Phi)[c(1:5,54:57),]
   (Intercept) flood:age0 flood:age1+
1            1          0           0
2            1          0           0
3            1          0           1
4            1          0           1
5            1          0           0
54           1          1           0
55           1          0           1
56           1          0           0
57           1          0           0
```

In non-flood years, survival is determined by β_1 for both age 0 and 1+, but in years with flooding the link for survival is $\beta_1 + \beta_2$ and for age 1+ it is $\beta_1 + \beta_3$. Now, if I thought that only age0 would be impacted by flooding and adult survival would be the same as in non-flood years, that model could be constructed by creating another 0/1 dummy variable which is 1 when age=0 and using it and the flood variable as a product or by simply creating a 0/1 dummy variable which is 1 for age=0 and flood=1. Either of the following approaches will create the same design matrix.

Approach 1:

```
> newdipper.ddl$Phi$young=0
> newdipper.ddl$Phi$young[newdipper.ddl$Phi$age=="0"]=1
> model.matrix(~flood:young,newdipper.ddl$Phi)[c(1:5,54:57),]
   (Intercept) flood:young
1            1           0
2            1           0
3            1           0
4            1           0
5            1           0
54           1           1
55           1           0
56           1           0
57           1           0
```

Approach 2:

```
> newdipper.ddl$Phi$young_flood=0
> newdipper.ddl$Phi$young_flood[newdipper.ddl$Phi$age=="0"&
                                newdipper.ddl$Phi$flood==1]=1
> model.matrix(~young_flood,newdipper.ddl$Phi)[c(1:5,54:57),]
   (Intercept) young_flood
1            1           0
2            1           0
3            1           0
4            1           0
5            1           0
54           1           1
55           1           0
56           1           0
57           1           0
```

You are essentially defining specific design matrix columns like you would in MARK except that with code it will fill in the proper places with 0s and 1s for you.

That brings us to the final topic of using individual covariates in formula and unlike the rest of the above material it is quite a bit easier to understand. However, I can't show you how they are used with `model.matrix` because individual covariates are not in the design data. There is no concept of an individual animal in the design matrix that is created with the design data. It is creating a design matrix for the parameters which are assumed to be the same for all animals in a particular group. The group factor variables do allow you to partition animals into groups but using individual covariates as a grouping factor isn't what you want to do because the individual covariates are to be treated as a numeric variable. So how does it work? If you are a MARK user you'll know that individual covariates are entered into the design matrix by including the name of the individual covariate into a position of the design matrix. Then when MARK computes the parameters for each animal, it substitutes the value of that individual covariate into the position containing the name of the covariate (e.g., `weight`). Thus, the actual design matrix changes for each animal.

Now there is no way that `model.matrix` was going to allow something like that because it works with data and not variable names. However, my valued friend the dummy variable came to the rescue! I was able to incorporate individual covariates in the formula by adding a dummy variable into the design data for each individual covariate. Then I used `model.matrix` to produce the design matrix and subsequently filled in all design matrix entries for the individual covariate with the name of the covariate. This allowed using individual covariates as additive effects and in interactions with other

individual covariates, with numeric design variables and with factor design variables. For interactions of individual covariates or individual covariates with numeric design variables (e.g., Time), RMark uses the product(var1,var2) entry that MARK recognizes. Interactions of individual covariates with factor variables simply result in multiple columns containing the name of the individual covariate where the dummy variable is 1 for the various levels of the factor variable. I'll show some of these examples below.

c. *What is PIM simplification and why should I care?*

For some capture-recapture studies with many occasions and many groups, the size of the design matrix becomes very large because RMark uses the all-different PIM formulation which has a different index for each parameter and each index requires a row in the design matrix. This has several consequences and none are good. As the size of the design matrix increases, for mark.exe the amount of computations and subsequently the length of time increase to fit models to the data. However, a more severe consequence is that mark.exe computes the variance-covariance matrix which has n^2 elements where n is the number of rows in the design matrix. If n gets very large, there may be insufficient memory to compute the real variance-covariance matrix and the model will fail to run.

This problem became apparent as RMark was expanded to include models with more than two types of parameters, like the live-dead models. For the newdipper data with age class and sex groups, there are 84 possible parameters for Phi and 84 for p, so there could be as many as 168 rows in the design matrix. That isn't an exceptionally large problem but, it is easy to create examples with 10,000 rows in the design matrix which would require 1GB of memory for the variance-covariance matrix. To solve the memory issue and to speed up computation it is useful to reduce the number of real parameters to the smallest number that are needed. That is the simplification process.

For example, with the newdipper data if the model was Phi(~1)p(~1) then Phi and p are constant and each of the 84 real Phis and 84 real ps have the same value, so we really only need 2 parameters. The PIM coding is simplified to 2 indices and the original indices of 1-84 for Phi are assigned to 1 and indices 85-168 for p are assigned to 2. We can see this in the model results as shown below. The design matrix that is used with MARK is a 2x2 identity matrix:

```
> mymodel=mark(newdipper.proc,newdipper.ddl,
               model.parameters=list(Phi=list(formula=~sex+age)),output=FALSE)
> mymodel$design.matrix
                                 Phi:(Intercept) p:(Intercept)
Phi gFemaleAdult c1990 a1 t1990 "1"             "0"
p gFemaleAdult c1990 a2 t1991   "0"             "1"
```

The original indices are saved as a vector containing the new index after simplification in the model element `simplify$pim.translation`:

```
> mymodel$simplify$pim.translation
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 2
 [86] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2

> table(mymodel$simplify$pim.translation)
```

```
 1  2
84 84
```

These original indices are saved to construct all of the 168 original parameter values. This is necessary to enable model averaging which requires averaging across all possible parameter values. As the models change, the number and indices of the simplified parameters change to match the model complexity, but the underlying all-different indices remain the same. The simplified parameters can also be examined with the PIMS function. They are quite simple for the ~1 model so we'll consider a more complicated model. There are 4 simplified parameters for Phi with one for each of the sex-age categories.

```
> PIMS(mymodel,"Phi")
group = sexFemale.classAdult
     1990 1991 1992 1993 1994 1995
1990    1    1    1    1    1    1
1991         1    1    1    1    1
1992              1    1    1    1
1993                   1    1    1
1994                        1    1
1995                             1
group = sexMale.classAdult
     1990 1991 1992 1993 1994 1995
1990    2    2    2    2    2    2
1991         2    2    2    2    2
1992              2    2    2    2
1993                   2    2    2
1994                        2    2
1995                             2
group = sexFemale.classHatch-year
     1990 1991 1992 1993 1994 1995
1990    3    1    1    1    1    1
1991         3    1    1    1    1
1992              3    1    1    1
1993                   3    1    1
1994                        3    1
1995                             3
group = sexMale.classHatch-year
     1990 1991 1992 1993 1994 1995
1990    4    2    2    2    2    2
1991         4    2    2    2    2
1992              4    2    2    2
1993                   4    2    2
1994                        4    2
1995                             4
```

The appropriate simplified index is stored in the vector of 84 all-different indices. For example, the simplified index 3 is stored in positions 43, 49, 54, 58, 61, and 63 (the all-different indices).

```
> PIMS(mymodel,"Phi",simplified=FALSE)
group = sexFemale.classAdult
     1990 1991 1992 1993 1994 1995
1990    1    2    3    4    5    6
1991         7    8    9   10   11
1992             12   13   14   15
1993                  16   17   18
1994                       19   20
1995                            21
group = sexMale.classAdult
     1990 1991 1992 1993 1994 1995
1990   22   23   24   25   26   27
1991        28   29   30   31   32
```

```
1992             33   34   35   36
1993                  37   38   39
1994                       40   41
1995                            42
group = sexFemale.classHatch-year
     1990 1991 1992 1993 1994 1995
1990   43   44   45   46   47   48
1991        49   50   51   52   53
1992             54   55   56   57
1993                  58   59   60
1994                       61   62
1995                            63
group = sexMale.classHatch-year
     1990 1991 1992 1993 1994 1995
1990   64   65   66   67   68   69
1991        70   71   72   73   74
1992             75   76   77   78
1993                  79   80   81
1994                       82   83
1995                            84
```

To some degree this is more than you really need to know because the code does all of the PIM translation for you. However, it is important to know that the simplification occurs because if you look at the MARK output, the labels of the parameters for the real parameters may no longer make sense. But if you look at the real parameters with RMark they are shown using the all-different indices with the proper labels and associated design data.

d. How do I construct formulas for my model?

Be careful. You are entering a thinking zone! Be sure to be wide awake or well-caffeinated or both. Clearly I can't tell you how to construct formulas for your model without knowing the particulars about your data and data collection and what you envision as the biological models for your data. Thus, you needn't be wide awake for this section but you do need to be wide awake when you develop your models. You need to think, think and then think some more. RMark makes it extremely easy to build models – almost too easy. Don't let RMark become a tool whereby you stop thinking about the models. RMark saves you time by creating the design matrices. Take that time and apply it to thinking about the models you are developing for your data. To do a good job, you need to understand the design data and how the formula creates design matrices and how these relate to the parameters. Everything we have covered so far! If you don't understand something, you will likely fail to build useful models. You need to understand the difference between a cohort effect and a time effect, how aging works in the design data and all the other topics covered here. The most difficult part is translating your ideas about the biology and data collection into design data and formula. Also keep in mind the following points:

1) Too many models can be a bad thing! That can cause paralysis of analysis and lead you to nonsensical "best" model.
2) A nonsensical "best" model is still nonsensical and not useful!

An easier question that I can address is "How do I know if I have an over-parameterized model"? This is probably the most common error in creating design matrices with formula in RMark. An over-parameterized model means you have extra parameters that are not needed. After MARK fits a model to the data, it "counts" the parameters and it

excludes any that are "singular". If there are no "singular" parameters then your model does not have extra unused parameters. If there are no singular parameters then results$singular will be empty. Also, the summary of the model will provide a parameter count and it will not be followed by (unadjusted=nn) which shows the count that MARK reported. However there are several reasons for parameters to be singular and you need to be able to distinguish between the reasons.

"Singular" parameters are parameters that 1) are confounded with another parameter in the model, or 2) they represent a data category which doesn't have any data, or 3) they are from a "duplicative" design matrix column, or 4) they end up at a boundary because that is the best estimate for that parameter. Unfortunately, MARK cannot distinguish the reasons so you need to interpret. Strictly you should only exclude those parameters in category 1 or 2 above. With over-parameterized models we are referring to categories 2 and 3 and this is often correctable because it can result from improperly specifying the formula.

It is important to recognize that RMark assumes that all of your beta parameters (number of columns for your design matrix) are estimable and not singular. Even if MARK reports a lower count, RMark will still use the complete count in calculation of AICc etc unless you tell it otherwise with the argument (`adjust=FALSE` in `mark`). I chose this approach because I believe that in many situations overly-complex models are fitted to data because the model contains too many parameters, the data are sparse and many of the parameters end up at boundaries like p=0 or Phi=1. Most of these parameters fall into category 4 and should be counted but are incorrectly excluded by MARK. The MARK interface provides the opportunity to adjust the parameter count upwards. I've simply taken the opposite approach with RMark by counting every beta parameter and then providing you with the opportunity to decrease the parameter count (function `adjust.parameter.count`) when appropriate. I believe this is a more conservative approach that guards against accidentally fitting overly-complex models that are not truly supported by the data. I believe it is better to over-estimate the number of parameters and end up with a simpler model that is supported by the data than choosing a model that is not truly supported by the data.

To identify confounded parameters you need to understand the specifics of your capture-recapture model. For example, with the CJS model if you fit Phi(t)p(t) (Phi=~time and p=~time), then only the product of the Phi*p is estimable for the last occasion. For example, with the dipper data there are 12 beta parameters but only 11 parameters are estimable. The help files in MARK will help you determine what your parameter count should be with specific types of models.

Parameters that are at a boundary are easy to identify because they usually have standard errors that are 10 or 100 times greater than the beta estimate. If all of the standard errors for the beta estimates for a particular parameter (eg Phi) or for a particular factor covariate are all large, then it is highly likely that you have specified a formula that has created extra and unneeded beta parameters. However, any beta parameter that has a large standard error should be investigated and you should make

sure you understand why it is happening and be able to rule out the possibility of an over-parameterized model.

To help clarify some of this I'll make some errors that cause the models to have extra beta parameters. One way this can happen is by creating factor variables that have extra levels that are not represented in the design data. For example, I'll use the newdipper example and I'll misclassify the times to include a time that is not in the design data which will incorrectly end up being the intercept in the model:

```
> summary(newdipper.ddl$p$time)
1991 1992 1993 1994 1995 1996
   4    8   12   16   20   24
> newdipper.proc=process.data(newdipper,model="CJS",begin.time=1990)
> newdipper.ddl=make.design.data(newdipper.proc)
> newdipper.ddl$p$timebin=cut(newdipper.ddl$p$Time+1991,
    c(1990,1991,1993,1994,1997),right=FALSE)
> summary(newdipper.ddl$p$timebin)
[1990,1991) [1991,1993) [1993,1994) [1994,1997)
          0           3           3          15
> mark(newdipper.proc,newdipper.ddl,
  model.parameters=list(p=list(formula=~timebin)))
```

In the output it reports that MARK only counted 4 of 5 specified beta parameters and if you look at the betas in the summary something is definitely wrong:

```
Note: only  4  parameters counted of  5  specified parameters
AICc and parameter count have been adjusted upward

Output summary for CJS model
Name : Phi(~1)p(~timebin)

Npar :  5  (unadjusted=4)
-2lnL:  665.357
AICc :  675.4999  (unadjusted=673.45202)

Beta
                      estimate          se           lcl          ucl
Phi:(Intercept)      0.2350760   0.1011064     0.0369074    0.4332447
p:(Intercept)        1.5853556 549.0628100 -1074.5778000 1077.7485000
p:timebin[1991,1993) 0.1011729 549.0647700 -1076.0658000 1076.2682000
p:timebin[1993,1994) 0.3374842 549.0625500 -1075.8251000 1076.5001000
p:timebin[1994,1997) 0.9467993 549.0621300 -1075.2150000 1077.1086000
```

All of the beta parameters for p have very large standard errors and the reason is that 1990 doesn't exist in the design data and the factor variable for timebin has it as the first level. Now if we were to relevel the timebin factor variable so [1992,1993) is the intercept then we get the following summary:

```
> newdipper.ddl$p$timebin=relevel(newdipper.ddl$p$timebin,"[1991,1993)")
> summary(newdipper.ddl$p$timebin)
[1991,1993) [1990,1991) [1993,1994) [1994,1997)
          3           0           3          15
> mark(newdipper.proc,newdipper.ddl,
                  model.parameters=list(p=list(formula=~timebin)))

Output summary for CJS model
Name : Phi(~1)p(~timebin)
```

```
Npar :  4
-2lnL:  665.357
AICc :  673.452

Beta
                      estimate        se        lcl       ucl
Phi:(Intercept)      0.2350760 0.1011064  0.0369073 0.4332446
p:(Intercept)        1.6865288 0.5634069  0.5822511 2.7908064
p:timebin[1993,1994) 0.2363103 0.8492690 -1.4282569 1.9008776
p:timebin[1994,1997) 0.8456262 0.7137895 -0.5534013 2.2446537
```

Hmm! How did that happen? RMark will drop extraneous factor levels but only if the extraneous level is not the first level for the factor variable. Moral of this story is to check your design data carefully and make sure it is defined properly.

Here is another possibly stupid example whereby you create a covariate that duplicates another covariate.

```
> newdipper.proc=process.data(newdipper,begin.time=1990,groups=c("sex","class"),
+                                          age.var=2,initial.age=c(1,0))
> newdipper.ddl=make.design.data(newdipper.proc,
+                   parameters=list(Phi=list(age.bins=c(0,1,6))),right=FALSE)
> levels(newdipper.ddl$Phi$age)=c("0","1+")
> newdipper.ddl$Phi$young=0
> newdipper.ddl$Phi$young[newdipper.ddl$Phi$age=="0"]=1
> mark(newdipper.proc,newdipper.ddl,
     model.parameters=list(Phi=list(formula=~age+young)))

Note: only  3  parameters counted of  4  specified parameters
AICc and parameter count have been adjusted upward

Output summary for CJS model
Name : Phi(~age + young)p(~1)

Npar :  4  (unadjusted=3)
-2lnL:  666.7571
AICc :  674.8521  (unadjusted=672.814)

Beta
                estimate          se         lcl        ucl
Phi:(Intercept) 0.2018604 144.0497800 -282.135710 282.539430
Phi:age1+       0.0198416 144.0497800 -282.317740 282.357420
Phi:young       0.0818271 144.0498500 -282.255890 282.419550
p:(Intercept)   2.2223134   0.3255009    1.584332   2.860295
```

Essentially I have artificially created 3 variables for 2 factor levels for age. Two morals to this story: 1) check your design data carefully and make sure it is defined properly, and 2) make sure you think about what you want in your design matrix and why! The above may look stupid but it can happen more subtlely with errors in specifying the design data.

Another way to introduce extra parameters is to forget to use ~-1+ age:time instead of ~age:time or forgetting to add remove.intercept=TRUE with ~sex+age:time. Whenever you use a ":" interaction of two factor variables you need to remove the intercept and if you don't the model contains an extra beta that is not needed.

Even though RMark provides a quick way to create design matrices, the best way to avoid over-parameterized models is to visualize and understand the design matrix that you are creating with any formula.

Third Lab Exercise: Scripting and running a complete analysis of the dipper data

1) *Use the code in ClassScript.r to modify myscript.r to import data, create an Adult and Hatch-year group in the data, bin ages to 0,1+ for Phi and p and add the following models:*
   a. *Phi: age*
   b. *Phi: age + sex*
   c. *Phi: age*sex*
   d. *p: age*
   e. *p: age + sex*
   f. *p:age+sex+time*
   g. *p:age+sex+Time*
2) *Examine the model selection table. What is the best model? Is it clearly better than all of the other models?*
3) *What is the interpretation of the beta parameters for Phi in the age*sex model? Use plogis to construct the real parameter estimates for the model with Phi(~age*sex) p(~Time) . Modify this model to get individual estimates for each age-sex class? How do they compare?*
4) *Get model.averaged estimates of p.*

X. Fixing real parameter values

a. *Entire parameter*

Occasionally it is necessary or appropriate to fix the values of one or more real parameters. There are several ways to do this in RMark such that the model in MARK will constrain the parameter(s) to the value(s) that you specify. The most typical way is to use the `fixed` argument in the parameter specification list.

To specify that each parameter of a particular type is to be fixed at the same value then you use `fixed=value` where `value` is the real parameter value that you specify. For example, to specify that every animal is seen (i.e., known-fate analysis), you could specify the model for p as follows:

```
p.fixed=list(formula=~1,fixed=1)
```

Another example is the live-dead models, where one may set the fidelity parameter (F) to one if captures and dead recovery areas are the same.

b. *Fixed by design data fields*

A slightly more general format for fixing subsets of the parameters can be used with the design data fields `time`, `cohort` or `age`. These are most useful when you are setting a subset of the parameters based on one of these fields to a specific example. A common example is setting p to a set of times. What if I didn't recapture in 1991 and 1995 and I wanted to set p=0 for those years. I could do that by setting `fixed` to a list as follows:

```
p.fixed=list(formula=~1,fixed=list(time=c(1991,1995),value=0))
```

That specifies a model in which p is the same for 1992-1994 and 1996 and is 0 for 1991 and 1995. We could also specify any other formula for the non-fixed p as well. For example,

```
p.time=list(formula=~time,fixed=list(time=c(1991,1995),value=0))
```

would have 4 separate parameters for 1992-1994 and 1996 and would set 1991 and 1995 to 0. In place of time you could also specify `cohort` or `age` and specify the list of cohorts or ages and a value:

```
p.time=list(formula=~time,fixed=list(cohort=c(1990,1992),value=0))
```

If the values are different for the different times, ages or cohorts then you might expect you could do as follows:

```
p.time=list(formula=~time,fixed=list(cohort=c(1990,1992),value=c(0,1)))
```

but you cannot and you will get an error saying the length of the indices and values do not match. You could deservedly scratch your head in wonder because each has 2 values, but the indices are the numbers for the parameters for p which have the value of cohort of 1990 and 1992 and there are far more than 2 of those. In fact there are 40 of them with 24 for 1990 and 16 for 1992:

```
> dim(newdipper.ddl$p[newdipper.ddl$p$cohort%in%c(1990,1992),])
[1] 40  9
> dim(newdipper.ddl$p[newdipper.ddl$p$cohort%in%c(1990),])
[1] 24  9
> dim(newdipper.ddl$p[newdipper.ddl$p$cohort%in%c(1992),])
[1] 16  9
```

What would work would be the following:

```
p.time=list(formula=~time,
            fixed=list(cohort=c(1990,1992),value=c(rep(0,24),rep(1,16))))
```

You can think of fixing parameters based on time, cohort or age as a short-hand way of saying find me all the indices with these values of time, cohort or age. You may also naturally try to replace time, cohort or age with some other default or user-defined design data field. Many have tried this and all have failed because the code is not written to recognize that approach. But there is a more general way to specify fixing parameters that will let you accomplish the same thing.

c. General approach to fixing parameters

Any set of parameters can be fixed to any set of values by specifying the set of real parameter indices and values for each of the parameters with those indices. These indices are the row numbers in the design data and not the all-different indices. Thus for p in the newdipper, the first index is 1 and not its PIM index of 85. By using the row numbers (names) of the design data, you can construct the set of indices easily by using

the row numbers that match your selection criterion for the design data. For example, our previous example could also be constructed as follows:

```
p.cohort.1990=as.numeric(row.names(newdipper.ddl$p[newdipper.ddl$p$cohort==1990,]))
p.cohort.1992=as.numeric(row.names(newdipper.ddl$p[newdipper.ddl$p$cohort==1992,]))
p.time=list(formula=~time,fixed=list(index=c(p.cohort.1990,p.cohort.1992),
            value=c(rep(0,length(p.cohort.1990)),rep(1,length(p.cohort.1992)))))
> p.time
$formula
~time

$fixed
$fixed$index
 [1]  1  2  3  4  5  6 22 23 24 25 26 27 43 44 45 46 47 48 64 65 66 67 68 69 12 13 14 15 33 34
35 36 54 55 56 57 75 76 77 78

$fixed$value
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

A slightly more complicated example may help to demonstrate the flexibility better. What if I wanted to fix p to be 0 for cohort 1990 in years 1992 and 1995 and for cohort 1992 in years 1993, 1994 and 1996.

```
p.cohort.1990=as.numeric(row.names(newdipper.ddl$p[newdipper.ddl$p$cohort==1990 &
newdipper.ddl$p$time%in%c(1992,1995),]))
p.cohort.1992=as.numeric(row.names(newdipper.ddl$p[newdipper.ddl$p$cohort==1992&
newdipper.ddl$p$time%in%c(1993,1994,1996),]))
p.time=list(formula=~time,
             fixed=list(index=c(p.cohort.1990,p.cohort.1992),value=0))
> p.time
$formula
~time

$fixed
$fixed$index
 [1]  2  5 23 26 44 47 65 68 12 13 15 33 34 36 54 55 57 75 76 78

$fixed$value
[1] 0
```

Any selection can be made and you can use the numeric value of the row.names to specify the indices. You can partition the selections and combine them or use a single more complicated selection from the design data.

d. Fixing parameters by removing design data

One other way to fix parameters is to remove the design data rows for parameters. This approach can only be used to fix parameters to a default that is not user-defined. Primarily this is used to set parameters to defaults when there was no data collection for a year or cohort and the defaults are chosen such that the missing data is properly accounted in the model. For example, if we only released newly tagged dippers every other year, then we would have no data for cohorts 1991, 1993, 1995 and there would be no way to estimate any cohort-specific parameters for these data. If we were to delete the rows in the p and Phi design data for these cohorts, it will automatically set Phi to be 1 and p to be 0 for those parameters. With p set to 0, having a count of 0 for recaptures for the cohort at each occasion does nothing because it adds 0 * log(0). Each type of parameter has a default value and if a row of the design data is deleted that parameter is fixed to its default value.

For the case, in which every other cohort was not released, this is detectable in the capture history and it can be handled automatically with the argument `remove.unused` in the function `make.design.data`. If this is set to TRUE then any unnecessary design data is removed automatically.

XI. Goodness of fit/overdispersion

a. release.gof

Goodness of fit is a topic that is not well-developed in RMark, so I've also included a section here that discusses exporting data/models to the MARK interface to use it as well. What RMark can do is to call RELEASE to compute Test2+Test3 chi-square tests for a CJS model to obtain an estimate $\hat{c}$ for overdispersion and it can use an estimate $\hat{c}$ of over-dispersion to adjust model selection critierion values and inflate standard errors and confidence intervals.

If appropriate, to get a RELEASE goodness-of-fit test, call process.data with model="CJS" and use release.gof with the processed data list. For example,

```
> newdipper.proc=process.data(newdipper, model="CJS")
> release.gof(newdipper.proc)
RELEASE NORMAL TERMINATION
      Chi.square df      P
TEST2     9.4797  4 0.0502
TEST3     3.7547  8 0.8786
Total    13.2343 12 0.3522
```

It is important to know that the global model is Phi(g*t)p(g*t) which means it assumes group-time dependence interaction for Phi and p and asking if that model provides a reasonable fit. In the above, I did not include any groups so the global model was Phi(t)p(t) and the conclusion is that the fit is adequate, so there would be no reason to adjust for over-dispersion. If I thought my global model was Phi(sex*t)p(sex*t), I would have done as follows:

```
> newdipper.proc=process.data(newdipper, model="CJS",groups="sex")
> release.gof(newdipper.proc)
RELEASE NORMAL TERMINATION
      Chi.square df      P
TEST2     7.5342  6 0.2743
TEST3    10.7735 15 0.7685
Total    18.3077 21 0.6295
```

As I expected, the fit improved (P increased) because as you split up the data into groups it is "estimating" more parameters and the goodness-of-fit improves. However, as the next example shows splitting the data too finely can result in non-intuitive results. The degrees of freedom for the test depend on the number of components of the test that have sufficient data. If you split the data too finely with groups, there can be fewer degrees of freedom and any strange deviations can cause the test to reject:

```
> newdipper.proc=process.data(newdipper,
model="CJS",groups=c("sex","class"),initial.age=c(1,0),age.var=2)
> release.gof(newdipper.proc)
```

```
RELEASE NORMAL TERMINATION
      Chi.square df      P
TEST2     5.3977  4 0.2489
TEST3    25.1616 12 0.0141
Total    30.5592 16 0.0153
```

Providing g*t as the global model has somewhat limited flexibility. In particular if you have age dependence in your parameters then you'll want to use initial age as a grouping variable as we did above. Time replaces age within a group of animals all of the same age. If all of your initial release cohorts are of the same age, then Test3 (cohort differences) will not exist.

b. *Exporting data/models to MARK*

Because the RELEASE goodness of fit test only applies to CJS model, has limited flexibility and the chi-square approximation can be suspect with many cells and little data, it is probably wiser to use the median c-hat approach in MARK. At present, to use that approach to estimate $\hat{c}$ for overdispersion, you'll need to export your data and global model to MARK. This is much less painful than using the MARK interface to build models because the interface is designed to import models from the output files which RMark saves. You do need to create the project with the MARK interface which means that you need to enter type of model, number of occasions, time intervals, number of groups and labels and any individual covariates. All of these quantities are stored in the data or processed data list in RMark. The steps you'll take are as follows:

1) Use export.chdata with your processed data list and it will create a .inp file for you.
2) Start MARK and create a new project and enter the information and use the .inp file created above as the encounter histories file name.
3) In RMark, use export.model to export the models you want to import into the MARK interface.
4) In MARK, click on Browse in the menu and Output should appear on the menu if it is not already there.
5) Select from the menu Output/Append. It will list a set of marknnnY.out and other files. You only want to select the marknnnY.out files to be appended and it will bring in all of the files associated with those output files.
6) Once the models have been appended, you can use them as you see fit in MARK.

There is one caution here. You have to realize that due to simplification, each of these models will have a different PIM coding so you cannot use model averaging like you would want. That would also be true if your models were built with the MARK interface and you used models with a mix of PIM codings. The other facilities should work fine like median c-hat.

c. *adjust.chat*

Once you obtain an estimate $\hat{c}$ for overdispersion you can use the function `adjust.chat` to set the value of $\hat{c}$ for a single model or for a marklist of models. By setting the value, it will adjust AICc values which will become QAICc values. Also, it will

inflate standard errors and confidence intervals for any real parameters that are computed within RMark (note: this may not work with the sin link at present). However, it will not change the values that are already in the MARK output files.

XII. Miscellaneous topics

a. Convergence and starting values

MARK fits models using numerical optimization code and while it is fairly reliable, you never have a 100% guarantee that it will find the parameters for the maximum of the likelihood function. For models like CJS, it is usually very reliable but if you branch out into multistrata models that reliability is lessened due to the type of model. You can help yourself and the code by checking for convergence and providing starting values. You can check for convergence by examining the log-likelihood values of nested models. If one model has k parameters and another model has those k parameters plus other parameters, then the larger model should have a negative log-likelihood that is smaller (if negative, more negative) than the simpler model. If that is not the case, then the more complex model did not converge.

A quick way to compare the results of models requires knowing a little more about `model.table`. The model table you see by typing the name of a marklist does not show all of the fields contained in model.table. Each `model.table` also contains a column for each type of parameter in the model (e.g., Phi and p for CJS):

```
> names(myresults$model.table)
[1] "Phi"       "p"         "model"     "npar"      "AICc"
"DeltaAICc" "weight"    "Deviance"
```

The contents of Phi and p are the formula for that model parameter. While model selection results are often shown in ascending order of DeltaAICc, sometimes it is also useful to see specific values shown as a matrix say with Phi formula values as the rows and p formula values as the columns. Using our earlier results with the newdipper data we can get all the AICc values in a matrix form:

```
> with(myresults$model.table, tapply(AICc, list(Phi,p), unique))
                    ~1    ~time    ~Time
~1            670.8660 678.7481 670.8170
~sex          672.7331 680.6496 672.6852
~sex + weight 672.0216 679.9837 671.9775
```

However, to look at nested models we want the -2Log-Likelihood values:

```
> with(myresults$model.table, tapply(Neg2LnL, list(Phi,p), unique))
                    ~1    ~time    ~Time
~1            666.8377 664.4802 664.7601
~sex          666.6762 664.3043 664.5902
~sex + weight 663.9265 661.5510 661.8347
```

In each case above the more complex model has a smaller Neg2LnL value.

Convergence can be improved by providing good starting values. Also, if you can provide starting values that are close to the final estimates, the time required to fit models will be reduced and sometimes substantially. With the `initial` argument, you can specify initial values for the beta parameters to the `mark` or `mark.wrapper` functions as a vector of values for the model but the values must be in the correct order and the correct length. An easier way is to use an initial model fit for specifying initial values of a sequence of models. The code will find any matching names of the beta parameters that are in common between the initial model and each new model and will use those as the starting values. For any beta that doesn't match, it will use 0 as the starting value. So for the example, of nested simple and complex models, you could use the simple model for starting values of the more complex model and it will assign the k starting values that are the same in both model and it will set all of the other parameter starting values in the complex model to 0. This will guarantee that it will have a negative log-likelihood at least as small as the simpler model because as long as the models are nested the simpler model is just the more complex model with the extra parameters set to 0.

A useful protocol is to fit a model with all of the covariates you are going to consider in an additive model for each of the parameters. Then use this model for the initial model for your function that fits all of the models. Along this same line, models with individual covariates take much longer to run than models without them. Thus, I recommend fitting a sequence of models without individual covariates and seeing if you can eliminate any of the candidate models. Then when you have a final sequence, use your most complex model without individual covariates for starting values of the models with individual covariates. This should save a considerable amount of computing time.

b. Time-varying individual covariates

As mentioned earlier, time-varying individual covariates should not be confused with occasion-dependent covariates (e.g. weather, effort etc.). In this case we are talking about a covariate that is measured for each animal and is known at each time. They are very rare because you typically don't know these values without capturing the animal on each occasion. One exception is age which is handled with the design data but it could be handled with a time-varying individual covariate. Another exception is a "trap-dependence" covariate. One way to model "trap-happy" or "trap-shy" behavior is to model the capture probability of occasion k+1 based on what happened at occasion k. Note that if this is a true CJS model in which the animals are released then you would want to exclude the initial release. Anyhow, this is an example in which you know the value for each animal at occasions 2…k because it is the capture history value at occasion 1…k-1. Time-varying individual covariates are more common for occupancy models because the individuals are sites and the occasions are the times you visited the site and it is quite easy to imagine site-time specific covariates that might affect either occupancy of the site or the probability of observing animals at an occupied site.

Because the covariate values can vary by time, you need to have a covariate for each occasion and a value for each animal (site). The only trick to using time-varying individual covariates is to name them properly. To link the covariates to the occasion (time), they should be named with a common prefix (e.g., cov) and the suffix should be the label given to each occasion (time). The label depends on the value you assign to `begin.time` in

`process.data`, so you have to be consistent between labeling and the names of the covariate in your data. For example, let's assume that with newdipper the first capture is not a release and we want to model trap dependence for recapture probability for occasions 2 to 7. I know that my first occasion is 1990, so my recapture times will be labeled 1991 to 1996 and I'll want to label my variables td1991, td1992, …, td1996. When I create a formula, I'll use ~td and RMark will recognize that td is not a covariate or design data and it will use the names td1991, …, td1996 in the design data for the appropriate rows in the design data as they correspond to the particular times. An example of this is given in C.16 in Appendix C of the online book. Note that you should not use another variable in the design data or as an individual covariate that has the value td (or whatever you use as the prefix) or you will confuse RMark.

c. *Derived "parameters"*
For some models, MARK will derive "parameters" which are not parameters in the fitted model but are derived from the fitted parameters. For example, with the Pradel version of the Jolly-Seber models, the likelihood does not include N but it can be derived from the model parameters; whereas, with the Burnham version of the Jolly-Seber model N is one of the model parameters and is estimated directly. The derived quantities that are computed by MARK are extracted and the estimates are saved in `model$results$derived` and their variance-covariance matrix is in `model$results$derived.vcv`. Unfortunately, at this point they are not labeled in RMark so you'll have to examine the MARK output file if there is more than one derived quantity.

d. *Large analyses/insufficient memory/external storage*
R keeps the entire workspace in memory and if workspace size exceeds the available memory on your computer, you won't be able to use the workspace. When RMark was first developed the plan was to keep the input, output, vcv and res file contents in the workspace. It became clear quickly that was not possible, so they are stored in the same directory as the workspace. The compromise was to extract the results and only keep the design data and design matrix in the mark object which is stored in the workspace. However, with some large data sets with many complex models, even this becomes too large for the available memory on some machines. A solution for this case is to use `external=TRUE` in the call to `mark` or `mark.wrapper` and the contents of the mark object are stored in a saved image file (*.rda) in the directory with the output files. The mark object will contain the filename (e.g., mark001.rda) for the saved image; however, the mark object will act like any other mark object because the code will automatically load the content whenever it is used. If you need to save externally, you should review the help on `store` and `restore` which allow storing and restoring to and from external storage. You can also type `load(mymodel)` where `mymodel` is the name of a single model that has been saved externally. This will create an object called model in your workspace that is the `mark` object.

A related issue is simulation. If you are using R to simulate data and then calling MARK via RMark, you should be aware of the `delete=TRUE` argument for `mark` or `mark.wrapper`. It will automatically delete the input (*.inp) and MARK output files

(*.out, *.vcv, *.res) after the results have been extracted. This prevents cluttering your directory with possibly thousands of files.

e. Random effects/variance components

You can estimate variance components with a random effects model by exporting the model to MARK and choosing that option from the output model or you can use the function `var.components` in RMark. See `?var.components` for a description. If you use `var.components` the calculations are done in R and not with MARK. For the time being I suggest using MARK or both approaches to make sure the results with RMark are the same. I and others have done some testing but recently someone pointed out some discrepancies and I've not yet worked out the issue. If you use MARK for variance components, you'll want to use the simplified parameter indices which you can extract with the `PIMS` function for some models.

f. Profile likelihood confidence intervals

You can request that MARK construct profile confidence intervals by using the argument `profile.int=TRUE` with `mark` or `mark.wrapper`. Note that these are computed only by MARK and not by RMark so if you use `adjust.chat` after fitting the model or compute real parameter values at covariate values (e.g., `covariate.predictions`), the intervals are no longer profile intervals. You can specify a value for $\hat{c}$ with the `chat` argument of `mark` or `mark.wrapper` such that MARK will use that value in its calculation of the profile interval.

g. Delta method

Occasionally you'll want to compute your own derived quantities and you will want to compute a measure of precision and confidence interval. Each mark object contains the beta parameters (`results$beta`) and its variance-covariance matrix (`results$beta.vcv`) and it contains real parameter estimates and you can get the variance-covariance matrix for the real parameters from MARK (if use `realvcv=TRUE`) or with RMark functions like `covariate.predictions`. Thus, you need to decide if you are going to derive the quantity of interest from the beta parameters or the real parameters and then use the appropriate estimates and variance-covariance matrix.

The delta method is a standard approach for deriving measures of precision (variance = standard error^2) and covariances. It requires first derivatives of the derived quantity with respect to the parameter(s) and it needs a variance-covariance matrix (or just a variance if only one parameter is used). The first derivatives can either be computed numerically or preferably analytically. In the R package `msm`, there is a function named `deltamethod` which is given a formula for the derived quantity, the estimate(s) of the values used in the derived quantity and a variance-covariance matrix for the estimates. It creates an analytical derivative for the formula for the derived quantity and computes either the delta method standard error(s) or the variance-covariance matrix (`ses=FALSE`). An example using this package is provided in C.22 of Appendix C of the online book. In addition, a function `deltamethod.special` is provided with RMark to create the formula for special cases that often arise with capture-recapture analysis.

Open Lab: Work with individual data sets