

RMark Workshop

Jeff Laake

1 May 2015

1 What are MARK and RMark?

1.1 What is MARK and what does it do?

MARK is freely available software that was written by Dr. Gary White at Colorado State University and can be installed from the Colorado State University site (<http://welcome.warnercnr.colostate.edu/~gwhite/mark/mark.htm>) or from the Cornell site (<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. At this writing there are 155 types of models that MARK can analyze.

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.

1.2 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.

RMark DOES NOT fit models to data! It uses MARK to do that. RMark builds models that MARK fits to the data.

1.3 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. Scripting has the added advantage that analysis can be embedded in L^AT_EX documents with Sweave/knitr or rmarkdown documents with knitr to create reproducible research documents.

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 (parameter index matrices) or DMs (design matrices) which will be described later. Also there is 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 does not 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. Unfortunately that option will not be open to users of Mac or Linux computers.

1.4 Supported models

At the time of writing, RMark supports 97 of the 155 models in MARK. Many of the models that are not supported by RMark are versions of the supported models with miss-identification of marks incorporated. Adding models is typically not very difficult so if you find an unsupported model you would like to use contact me. You can see what models are available in MARK and which are supported by RMark by running the MARK interface and selecting Help/Data Types. Also, there is a pdf called MarkModels.pdf in the RMark sub-directory of your R library. This is often located in the directory "C:\Program Files\R\R-

v.v.v\library\RMark” where v.v.v is the version of R that you are using. The MarkModels.pdf also has the names of each type of parameter in the model (e.g., Phi and p in CJS). Those parameter names are used to specify the model to RMark.

After learning to use RMark with one of type of model, 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.

2 MARK/RMark help, documentation, and example datasets

2.1 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 1000+ 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. These workshop notes contain most but not all of the material in Appendix C. It is worth your time to read Appendix C which also provides detailed examples for multi-state, known fate, nest survival and occupancy models. 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. I have organized these notes in this fashion. We will start with a basic understanding of the Cormack-Jolly-Seber (CJS) model and how you use the MARK interface. Then I’ll move on to describe the use of 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 RMark errors are described in C.23 of the appendix.

2.2 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”)`.

2.3 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 about RMark to the Program MARK - RMark 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. BUT, before you post, please search the forum as your question may have already been answered! Also, provide enough detail including the code and any errors interleaved with the code so we can see where the problem occurred. You can imagine that it is not very useful to get a post like, "I was analyzing my nest survival data and I got an error. Can someone help me!"

3 A Simple CJS Example

So let's get started. We'll consider a capture-recapture experiment in which animals were individually marked and released on 4 occasions. In our experiment the population is open and animals may die between occasions but we are not considering births or immigration because the model is only for events after the initial release. So these could be tagged fish being released at dams from a hatchery and we want to estimate their survival over time. This type of model is called Cormack-Jolly-Seber (CJS) which is the surnames of the 3 folks that developed this type of model.

On the 3 occasions following the initial capture, marked animals are recaptured and re-released. The data are represented by a capture history (ch) which is also called an encounter history. The values in the history are either 1 meaning the animal was caught (seen)/initially released or 0 meaning the animal was not caught(seen). Each position in the capture history represents the 4 occasions in order. For example, the history 1001 represents an animal that was marked on the first occasion and was released. It was not seen on occasions 2 and 3 but was seen on occasion 4. Likewise, the history 0110 was an animal first marked and released on occasion 2, seen on occasion 3 and then not seen on occasion 4. Note that the capture history 0001 is animals marked and released on occasion 4 but this provides no information for the CJS model and can be excluded. In closed capture or Jolly-Seber models those data are informative and are included in the data.

There are two types of parameters in this model: 1) apparent survival which is named Phi and represented by the Greek letter ϕ and 2) capture/recapture probability named and represented by the letter p. Now to complicate our example somewhat I'll assume that we have both males and females and they can be easily distinguished. One goal is to decide if survival varies by sex. Also we could consider the possibility that capture probability is sex-specific. We also want to know if survival is a function of the animal's weight at release.

We can represent the data in tabular format with rows being the release cohorts (1-3) and columns being the recapture occasions (2-4). I'm excluding release cohort 4 because it does not provide any information.

Release Occasion	Recapture Occasion		
	2	3	4
1	1	2	3
2		4	5
3			6

The first release cohort can potentially be recaptured on occasions 2,3 and 4. The second can only be recaptured on occasions 3 and 4 and the third has only one opportunity to be recaptured on occasion 4. I'll get to the meaning of the numbers in the table later. We can also represent the data in tabular format with respect to the survival process:

Release Occasion	Survival Interval		
	1 to 2	2 to 3	3 to 4
1	1	2	3
2		4	5
3			6

To be recaptured on occasion 2, 3 or 4, the animal has to survive until that occasion. The survival period will be denoted by the time at the beginning of the period (e.g. time=1 for survival in interval from time 1 to time 2); whereas the time for capture is the occasion time.

An animal released on occasion 1, has to survive until occasion 2 to be captured on occasion 2. If we use ϕ for survival and p for capture probability, then the probability an animal released at occasion 1 is captured at occasion 2 is $\phi_1 p_1$. The subscript index for ϕ and p is the number in the table cell. Likewise, the probability that an animal is first captured on occasion 3 after release on occasion 1 (ch=101.) is $\phi_1 \phi_2 (1 - p_1) p_2$ because it has to survive until occasion 3, be missed on occasion 2 and then captured on occasion 3. The probability of observing each capture history (ch) is constructed in this manner. For example, the $\text{Pr}(\text{ch}=0101) = \phi_4 \phi_5 (1 - p_4) p_5$. A slightly more difficult one is $\text{Pr}(\text{ch}=0110) = \phi_4 p_4 (\phi_5 (1 - p_5) + (1 - \phi_5))$ because there are two possible outcomes for the last position. It is a 0 because either the animal survived and was not caught or the animal died. While it is useful to understand how these probabilities are constructed, you will not need to construct them because MARK handles all of that for you.

All of the parameters (e.g., Phi and p) are combined into a single parameter vector θ and each has its own index (not necessarily unique). Below are the survival and capture indices where each are unique. These are called PIMs (parameter index matrices) in MARK terminology.

Release Occasion	Survival Interval		
	1 to 2	2 to 3	3 to 4
1	1	2	3
2		4	5
3			6

Release Occasion	Recapture Occasion		
	2	3	4
1	7	8	9
2		10	11
3			12

Using the single vector θ , now $\text{Pr}(\text{ch}=0101) = \theta_4 \theta_5 (1 - \theta_{10}) \theta_{11}$. Even though the indices are unique, the values they represent may not be unique, as I show below with design matrices.

Now in this CJS model, the 12 parameters cannot all be uniquely estimated so the parameters must be restricted (e.g., set equal) in some fashion. One way to do that is to use a different set of indices for the PIMs. For example, if you wanted a model with constant survival and constant capture probability you could create the following PIMs:

Release Occasion	Survival Interval		
	1 to 2	2 to 3	3 to 4
1	1	1	1
2		1	1
3			1

Release Occasion	Recapture Occasion		
	2	3	4
1	2	2	2
2		2	2
3			2

and θ would be a vector with 2 values.

Another way to construct the same model would be with a design matrix (DM). Consider the following DM which has a row for each index and two columns for the estimated parameters. The indices (rows) are for the real parameters and the columns are the estimated beta parameters. The real parameters are computed from the beta parameters.

Index	β_1	β_2
1	1	0
2	1	0
3	1	0
4	1	0
5	1	0
6	1	0
7	0	1
8	0	1
9	0	1
10	0	1
11	0	1
12	0	1

There are 2 estimated beta parameters: one for survival and one for capture probability. They are labeled β_1 and β_2 respectively and are called beta parameters in the MARK/RMark output. If X is the design matrix and β is the column vector of parameters then multiplying the matrix X and β represented by $X\beta$ will give the beta values for each real parameter index. In the above example, the multiplication simply assigns β_1 to indices 1-6 and β_2 to indices 7-12. Now let's consider an example in which ϕ is constant but all of the p 's differ. A DM for such a model might look like:

Index	β_1	β_2	β_3	β_4	β_5	β_6	β_7
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	0	0	0	0
4	1	0	0	0	0	0	0
5	1	0	0	0	0	0	0
6	1	0	0	0	0	0	0
7	0	1	0	0	0	0	0
8	0	1	1	0	0	0	0
9	0	1	0	1	0	0	0
10	0	1	0	0	1	0	0
11	0	1	0	0	0	1	0
12	0	1	0	0	0	0	1

Now the vector β has 7 values. The product $X\beta$ assigns β_1 to indices 1-6, β_2 to index 7, $\beta_2 + \beta_3$ to index 8, $\beta_2 + \beta_4$ to index 9, ..., $\beta_2 + \beta_7$ to index 12.

While you may not have seen a DM before you used them when you learned simple and multiple linear regression. For simple linear regression, you were probably presented with an equation like $y = a + bx + \epsilon$ where y is the dependent variable, x is the independent variable, a is the intercept, b is the slope and ϵ is the error term. The equation for the mean value (excluding error term) can be written as $Y = X\beta$ where Y is a column vector of the n dependent variable values, β is the column vector $\begin{bmatrix} a \\ b \end{bmatrix}$ and the design matrix X is:

Intercept	x
1	x_1
1	x_2
1	x_3
1	...
1	x_n

where x_i is the value of x for the i^{th} observation. In regression the $X\beta$ values are the mean for the dependent variable values ($y_i, i=1, n$). For the example above the $X\beta$ values are $a + bx_1, a + bx_2, \dots, a + bx_n$ which are

the predicted mean for the dependent values ($y_{i=1,n}$). However, with capture-recapture models the $X\beta$ values are used to compute real parameter values which are used in the model for the “dependent variable” which is the capture history.

Most real parameters θ in capture-recapture models are bounded like a survival probability which can have any real value between 0 and 1. $X\beta$ values are unbounded (i.e., can take any value from -infinity to +infinity) and to relate these to the θ parameters we use a link function which bounds the real parameters. A common link function is the logit link which bounds the real parameters between 0 and 1 for probabilities. It is used in logistic regression for the probability of a binomial/Bernoulli response variable. The logit link is defined such that $\log(\theta/(1-\theta)) = \beta$ and the inverse logit link is $\theta = (1 + \exp(-\beta))^{-1}$. You may have seen the following alternative form for the inverse logit link $\theta = \exp(\beta)/(1 + \exp(\beta))$. By dividing the numerator and denominator by $\exp(\beta)$ you will get $(1 + \exp(-\beta))^{-1}$. This latter formulation is more numerically stable when x gets large and is used in the `logis` function in R.

The general formula relating real parameters to beta parameters is $\theta = \text{link}^{-1}(X\beta)$ where link^{-1} represents the inverse link function. Another useful link is the log link which restricts a parameter like abundance to be greater than 0. The log link is used in Poisson regression. Another useful link is the mlogit link which restricts a set of probabilities to sum to 1 which is used for multinomial probabilities. The mlogit link is used for transition probabilities in multistate models and in the entry probability parameter for the POPAN model. In MARK, the default link for probabilities is the sin link; however it cannot be used with general DMs such as those with numeric covariates or an intercept formulation. Thus, RMark uses the logit link by default for probabilities. I give an example showing the usefulness of the sin link in section 6.6.

Notice that the different types of parameters (ϕ and p) do not share columns of the design matrix. In general, the complete DM is constructed by creating a DM for each parameter type and then pasting together the sub-matrices and adding zeros such that rows corresponding to one type of parameter only contain 0 in the columns for another type of parameter as shown in the example below with 3 different types of parameters:

Sex	β_1	β_2	β_3	β_4	β_5
F	1	0	0	0	0
M	1	1	0	0	0
F	0	0	1	0	0
M	0	0	1	1	0
F	0	0	0	0	1
M	0	0	0	0	1

This does not mean that you cannot use the same covariate for different types of parameters. For example, you may want to use sex as a predictor of both survival and capture probability. The above example shows that each type of parameter can be sex-specific. The restriction does mean that the sex-effect variable cannot be restricted to have the same value for each type of parameter. When the parameter types are similar like p , capture probability and c , recapture probability in closed models it can be useful to share the column in the design matrix which is demonstrated in section 6.9.

As an aside, the negative log-likelihood that is minimized by MARK is $-\ln L = \sum_{\omega} Pr(\omega|\beta)$ where ω is the set of observed capture histories and β is the vector of parameters. The structure of $Pr(\omega|\beta)$ depends on the type of capture-recapture model but you need not worry about that because MARK does all of that calculation for you and RMark will create the PIMs and DM for you from formulas. What you do need to understand is what the parameters represent, what models are relevant for each parameter for your data and how you construct formulas to represent those models.

We will get to that in due time but before we venture into an example, we need to consider 2 further complications to the PIMs and DMs. The first is factor variables which are referred to as groups in MARK terminology. You can think of these variables as something that splits your data into different strata or groups. We will use a single factor variable “sex” with values “Female” and “Male” but you can define groups based on more than one factor variable (e.g., sex and region). When you define a group variable the set of PIMs expands. Below is the set of PIMs for our example with Females and Males.

Female		Survival Interval		
Release	Occasion	1 to 2	2 to 3	3 to 4
	1	1	2	3
	2		4	5
	3			6

Male		Survival Interval		
Release	Occasion	1 to 2	2 to 3	3 to 4
	1	7	8	9
	2		10	11
	3			12

Female		Recapture Occasion		
Release	Occasion	2	3	4
	1	13	14	15
	2		16	17
	3			18

Male		Recapture Occasion		
Release	Occasion	2	3	4
	1	19	20	21
	2		22	23
	3			24

Now we have twice as many possible parameters with 2 groups. If we used group variables with 10 values then we would have 120 possible parameters. If you imagine a scenario with many more occasions and many groups, you can see how managing the parameters with a GUI could become unmanageable. With 24 real parameters we will have 24 rows in the DM but we would still only have 2 columns if we chose to have a model with constant survival and constant capture probability.

One final complication with the DM is an individual covariate. This is a numeric covariate (**not** a factor variable) which can be different for each animal. For the time being we will only consider an individual covariate that is treated as static throughout the course of the experiment. Time-varying individual covariates are discussed in section 6.8. We will use the weight of the animal at the time of its initial release. With the 2 groups, a DM for a model with constant capture probability and survival that is dependent on **weight** would look as follows with only the beginning and end of the DM shown:

Index	β_1	β_2	β_3
1	1	weight	0
2	1	weight	0
3	1	weight	0
4	1	weight	0
5	1	weight	0
...
19	0	0	1
20	0	0	1
21	0	0	1
22	0	0	1
23	0	0	1
24	0	0	1

The important thing to note here is that the DM contains the name of the individual covariate; whereas factor (group) variables are handled by expanding the set of PIMs. That affects what you get and what you can do with these covariates in RMark. With the above DM and the logit link, the formula for survival probability is $(1 + \exp(-\beta_1 - \beta_2 \text{Weight}))^{-1}$ and the formula for capture probability is $(1 + \exp(-\beta_3))^{-1}$.

4 Analyzing the Simple CJS Example with MARK

The simulated data for our simple example is in 2 files. The first is `example.inp` which is in the format required for MARK and we will use it with the MARK GUI. The second is `example.txt` which is in the format for RMark. The MARK format can be converted to the RMark format with the function `convert.inp`. Our example data contains 600 capture histories with 300 females and 300 males. The experiment has 4 occasions each separated by a unit time interval (e.g., one year). On each of the first 3 occasions 100 males and 100 females were released and then possibly captured on subsequent occasions. The `ch` field in both files is the capture history which is a character string of length 4 with values 0 and 1 (e.g., 1001). Each file also has the weight value for the animal. The column is named “weight” in `example.txt` but is unnamed and is the last field in `example.inp`. The file formats differ in two ways:

- For specification of groups (factor variables), the RMark format specifies the value(s) of the variables used for groups; whereas, with the MARK format, the groups are denoted by a value of the frequency field (number of animals with that capture history) in different columns with one column per group
- Fields in `example.txt` are tab-delimited whereas in `example.inp` the fields are space delimited and each line ends with a ;.

```
MARKData=read.table(file="example.inp",header=FALSE,
  colClasses=c("character",rep("numeric",2),"character"),
  col.names=c("ch","Females","Males","Weight"))
head(MARKData)

##      ch Females Males Weight
## 1 0011         1     0  2.96;
## 2 0010         1     0  3.09;
## 3 0010         1     0  3.14;
## 4 0010         1     0  3.23;
## 5 0011         1     0  3.47;
## 6 0011         1     0   3.5;

RMarkData=read.table(file="example.txt",header=TRUE,
  colClasses=c("character","factor","numeric"))
head(RMarkData)

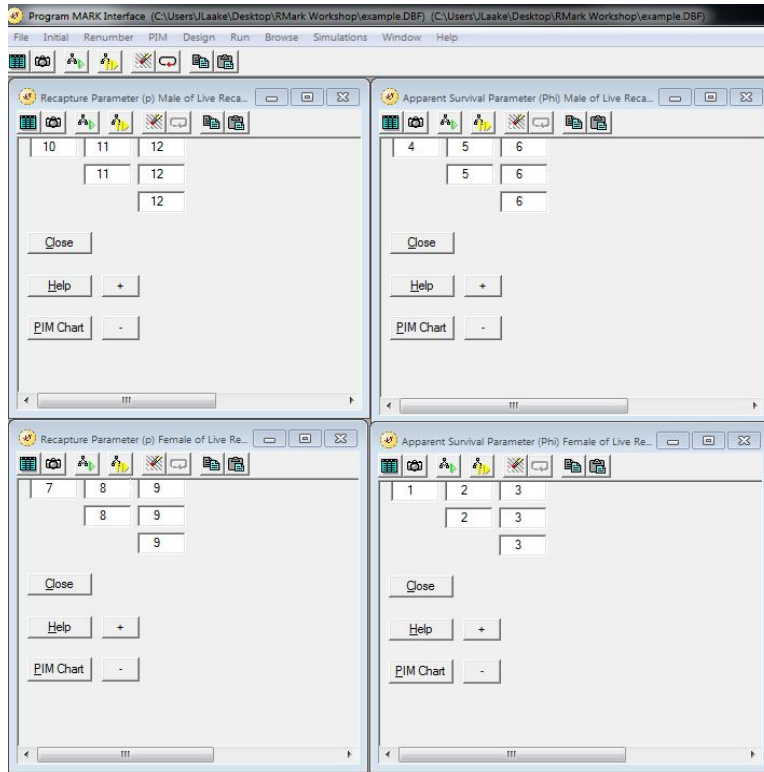
##      ch      sex weight
## 1 0011 Female   2.96
## 2 0010 Female   3.09
## 3 0010 Female   3.14
## 4 0010 Female   3.23
## 5 0011 Female   3.47
## 6 0011 Female   3.50
```

We will start by using MARK so you can understand what RMark must do to create models for MARK and also to compare the processes for analyzing data. The steps you'll take with MARK are as follows:

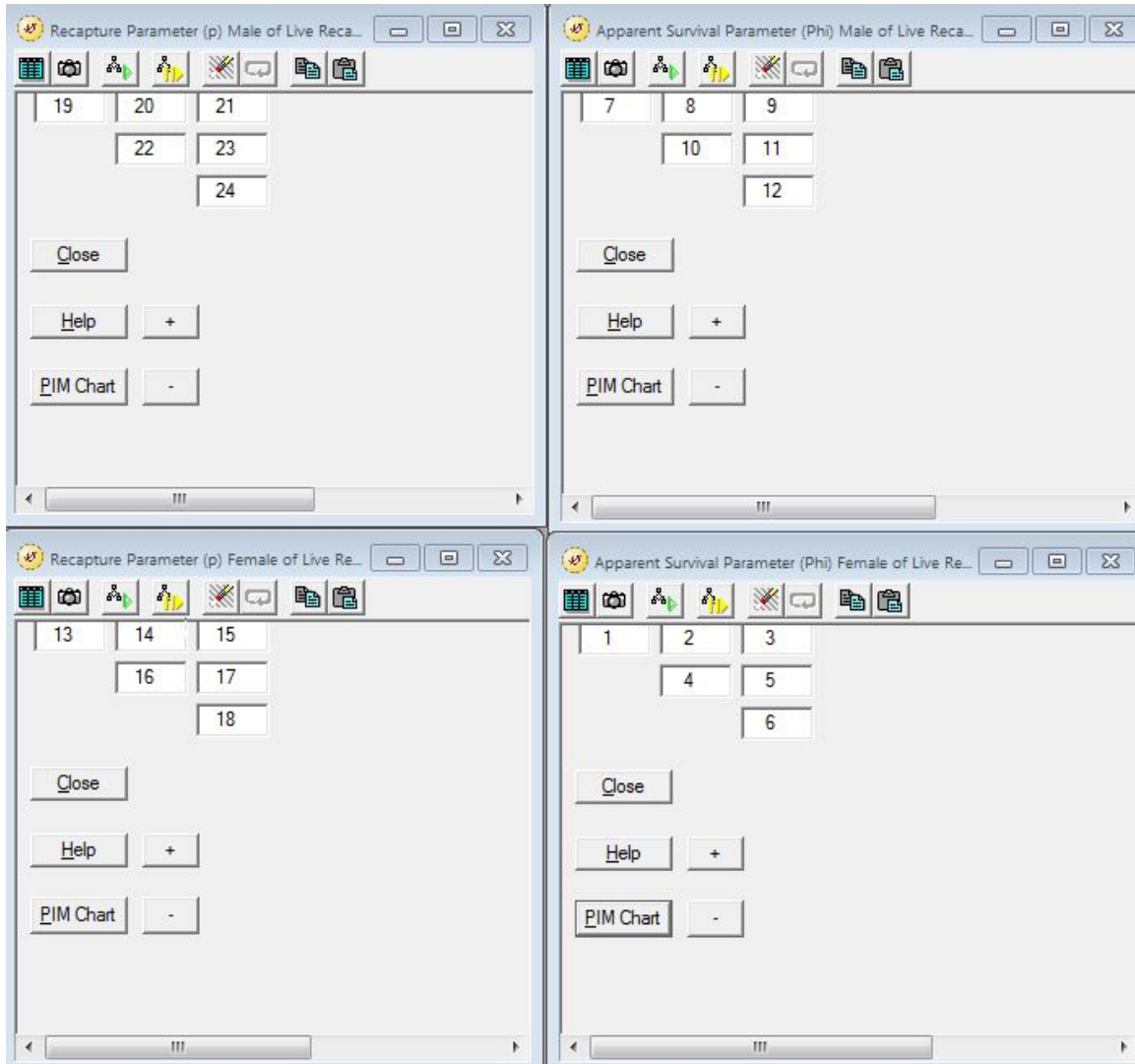
1. Start MARK interface program and select File/New.
2. Complete the form by:
 - (a) selecting the Data Type which for this example is Live Recaptures (CJS)
 - (b) Provide a title for the analysis (optional)
 - (c) Select the input file (`example.inp`); a results file name is automatically provided
 - (d) Specify the number of encounter occasions which is 4 for this example. We do not need to set the time interval lengths between occasions because they are all 1 for our example.

- (e) Specify the number of attribute groups which will be 2 and assign group labels which are “Female” and “Male”. The order must match the order of the columns in the input file.
- (f) Specify the number of individual covariates and their name(s). For our example, we have 1 individual covariate named weight
- (g) While not relevant for this example, for multistate models you will need to set the number and labels for states and for heterogeneity models the number of mixture distributions.
- (h) Once the form is completed as shown below you should press Ok.

After pressing OK, the Menu at the top of the screen will change and a window will open showing the PIM for survival for the female group. Select the PIM menu item, select all of the PIMS and press ok. If you then select Window/Tile you should see the following:



These are called time PIMS because they are setup with a different index for each time for each group. With this PIM structure you cannot have a model that has different real parameter values for each release cohort (row) or by age which are diagonals. If you want to fit models with cohort or age effects you need to expand the set of indices. The easiest way to change the PIM type is to use the PIM chart which shows all of the indices by group. By right clicking on each set you can select **All different** to make each index unique. Once you do that for each group you need to select **Renumber with no overlap** so each set of PIMs are different for capture and survival for each sex. If done correctly each blue block will be separate and there will be 24 indices. Once you close the PIM chart, the PIMs should look like the following as described in section 3:



It is possible to fit pre-defined models with MARK but my goal here is to show how you specify a model with the design matrix to compare with the RMark process of building models. To specify a model, select **Design/Reduced** from the menu and then specify 7 columns. The default is 24 which is also the number of rows and it could be used to specify an identity design matrix but not all of the parameters are estimable. The number of columns is the number of beta parameters. We are going to specify a model for survival that varies by sex and initial weight and capture probability varies by time and an additive sex effect. We will have 3 beta parameters (columns) for survival and 4 parameters for capture probability. How do I know that? For survival we need an intercept parameter for each sex and a slope for weight. For capture probability we need a parameter for each time plus one more for the additive sex effect. If you guess the number of columns incorrectly you can add or delete columns in the design matrix. After you press Ok, you will see a window that looks like the following:

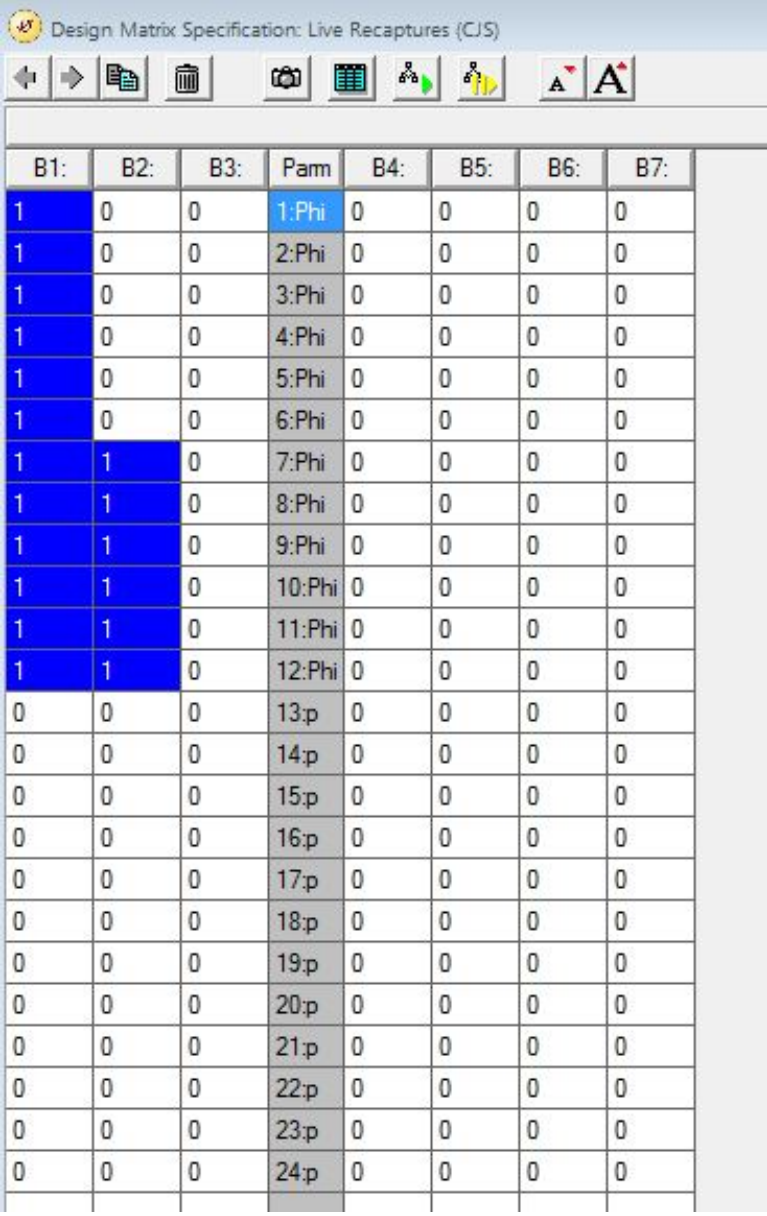
Design Matrix Specification: Live Recaptures (CJS)							
B1:	B2:	B3:	Parm	B4:	B5:	B6:	B7:
0	0	0	1:Phi	0	0	0	0
0	0	0	2:Phi	0	0	0	0
0	0	0	3:Phi	0	0	0	0
0	0	0	4:Phi	0	0	0	0
0	0	0	5:Phi	0	0	0	0
0	0	0	6:Phi	0	0	0	0
0	0	0	7:Phi	0	0	0	0
0	0	0	8:Phi	0	0	0	0
0	0	0	9:Phi	0	0	0	0
0	0	0	10:Phi	0	0	0	0
0	0	0	11:Phi	0	0	0	0
0	0	0	12:Phi	0	0	0	0
0	0	0	13:p	0	0	0	0
0	0	0	14:p	0	0	0	0
0	0	0	15:p	0	0	0	0
0	0	0	16:p	0	0	0	0
0	0	0	17:p	0	0	0	0
0	0	0	18:p	0	0	0	0
0	0	0	19:p	0	0	0	0
0	0	0	20:p	0	0	0	0
0	0	0	21:p	0	0	0	0
0	0	0	22:p	0	0	0	0
0	0	0	23:p	0	0	0	0
0	0	0	24:p	0	0	0	0

The rows of the DM are labeled with the parameter name and the index. There are 24 indices and thus 24 rows. What you put into the DM in a row should match what that index represents. For example, rows 13-18 represent female capture probabilities and thus should be used for the female parameters.

There are many different DMs that can be specified to represent the same model. With different DMs the beta parameters may differ but the real parameters and the resulting likelihood value should be the same. That said, the optimization is a numerical process and I have seen differences occur (rarely) because of differences in the DM due to the numerical algorithm not converging properly with sparse data. To understand DM construction, you need to understand the concept of contrasts. For example, I could specify the intercept for male and female survival with separate parameters or I could specify a single intercept and a second column which is the amount that males differ from females. There are two columns in both circumstances but the interpretation of the betas are different. I'm not sure if there is a name for the first approach but it could be called an identity contrast because each level of the factor variable has its own

separate intercept. The second approach is called a “treatment contrast” from the experimental concept that the intercept is for the control and the treatment effect is the difference created by the treatment. It needn’t be a control-treatment experiment, but one level of the factor is used as the intercept and the parameters for the remaining levels represent the difference from the intercept. With pre-defined models in MARK, it uses the last level as the intercept which is a SAS convention; whereas, in R, the first level is used as the intercept. The first level is defined by the alphanumeric ordering of the factor level names (e.g., Female and then Male) but you can change the ordering with the `relevel` function. Less commonly used contrasts in R are the sum and Helmert contrasts. I’ll focus on the first two approaches.

Let’s first use a treatment contrast for survival. Our model is $\sim \text{sex} + \text{weight}$ where the \sim means formula and the $+$ means an additive effect. We will make females the intercept and the sex effect will be how much males differ. We do that by adding a 1 in the first column for rows 1-12 and a 1 in column 2 for rows 7-12 which are for males as shown below:



B1:	B2:	B3:	Pam	B4:	B5:	B6:	B7:
1	0	0	1:Phi	0	0	0	0
1	0	0	2:Phi	0	0	0	0
1	0	0	3:Phi	0	0	0	0
1	0	0	4:Phi	0	0	0	0
1	0	0	5:Phi	0	0	0	0
1	0	0	6:Phi	0	0	0	0
1	1	0	7:Phi	0	0	0	0
1	1	0	8:Phi	0	0	0	0
1	1	0	9:Phi	0	0	0	0
1	1	0	10:Phi	0	0	0	0
1	1	0	11:Phi	0	0	0	0
1	1	0	12:Phi	0	0	0	0
0	0	0	13:p	0	0	0	0
0	0	0	14:p	0	0	0	0
0	0	0	15:p	0	0	0	0
0	0	0	16:p	0	0	0	0
0	0	0	17:p	0	0	0	0
0	0	0	18:p	0	0	0	0
0	0	0	19:p	0	0	0	0
0	0	0	20:p	0	0	0	0
0	0	0	21:p	0	0	0	0
0	0	0	22:p	0	0	0	0
0	0	0	23:p	0	0	0	0
0	0	0	24:p	0	0	0	0

Next we want to add an individual weight covariate which we do by entering the name “weight” in the column as shown below:

Design Matrix Specification: Live Recaptures (CJS)

B1:	B2:	B3:	Parm	B4:	B5:	B6:	B7:
1	0	weight	1:Phi	0	0	0	0
1	0	weight	2:Phi	0	0	0	0
1	0	weight	3:Phi	0	0	0	0
1	0	weight	4:Phi	0	0	0	0
1	0	weight	5:Phi	0	0	0	0
1	0	weight	6:Phi	0	0	0	0
1	1	weight	7:Phi	0	0	0	0
1	1	weight	8:Phi	0	0	0	0
1	1	weight	9:Phi	0	0	0	0
1	1	weight	10:Phi	0	0	0	0
1	1	weight	11:Phi	0	0	0	0
1	1	weight	12:Phi	0	0	0	0
0	0	0	13:p	0	0	0	0
0	0	0	14:p	0	0	0	0
0	0	0	15:p	0	0	0	0
0	0	0	16:p	0	0	0	0
0	0	0	17:p	0	0	0	0
0	0	0	18:p	0	0	0	0
0	0	0	19:p	0	0	0	0
0	0	0	20:p	0	0	0	0
0	0	0	21:p	0	0	0	0
0	0	0	22:p	0	0	0	0
0	0	0	23:p	0	0	0	0
0	0	0	24:p	0	0	0	0

Next we add the design for p by creating an intercept in column 4 which is the value of females at occasion 2 which is the first recapture occasion.

Design Matrix Specification: Live Recaptures (CJS)							
B1:	B2:	B3:	Parm	B4:	B5:	B6:	B7:
1	0	weight	1:Phi	0	0	0	0
1	0	weight	2:Phi	0	0	0	0
1	0	weight	3:Phi	0	0	0	0
1	0	weight	4:Phi	0	0	0	0
1	0	weight	5:Phi	0	0	0	0
1	0	weight	6:Phi	0	0	0	0
1	1	weight	7:Phi	0	0	0	0
1	1	weight	8:Phi	0	0	0	0
1	1	weight	9:Phi	0	0	0	0
1	1	weight	10:Phi	0	0	0	0
1	1	weight	11:Phi	0	0	0	0
1	1	weight	12:Phi	0	0	0	0
0	0	0	13:p	1	0	0	0
0	0	0	14:p	1	0	0	0
0	0	0	15:p	1	0	0	0
0	0	0	16:p	1	0	0	0
0	0	0	17:p	1	0	0	0
0	0	0	18:p	1	0	0	0
0	0	0	19:p	1	1	0	0
0	0	0	20:p	1	1	0	0
0	0	0	21:p	1	1	0	0
0	0	0	22:p	1	1	0	0
0	0	0	23:p	1	1	0	0
0	0	0	24:p	1	1	0	0

We want capture probability to vary by time so we need the last 2 columns for occasions 3 and 4. To fill these in you have to pay attention to which index (row) belongs to which time (occasion) in the PIMs. Indices 13 and 19 are for occasion 2, indices 14,16,20,22 are for occasion 3 and the remainder are for occasion 4.

Design Matrix Specification: Live Recaptures (CJS)

B1:	B2:	B3:	Parm	B4:	B5:	B6:	B7:
1	0	weight	1:Phi	0	0	0	0
1	0	weight	2:Phi	0	0	0	0
1	0	weight	3:Phi	0	0	0	0
1	0	weight	4:Phi	0	0	0	0
1	0	weight	5:Phi	0	0	0	0
1	0	weight	6:Phi	0	0	0	0
1	1	weight	7:Phi	0	0	0	0
1	1	weight	8:Phi	0	0	0	0
1	1	weight	9:Phi	0	0	0	0
1	1	weight	10:Phi	0	0	0	0
1	1	weight	11:Phi	0	0	0	0
1	1	weight	12:Phi	0	0	0	0
0	0	0	13:p	1	0	0	0
0	0	0	14:p	1	0	1	0
0	0	0	15:p	1	0	0	1
0	0	0	16:p	1	0	1	0
0	0	0	17:p	1	0	0	1
0	0	0	18:p	1	0	0	1
0	0	0	19:p	1	1	0	0
0	0	0	20:p	1	1	1	0
0	0	0	21:p	1	1	0	1
0	0	0	22:p	1	1	1	0
0	0	0	23:p	1	1	0	1
0	0	0	24:p	1	1	0	1

You can label each column so the labels appear in the output but we'll skip that step. You can always tell which index represents the intercept because it will only have a single 1 in the row and possibly individual covariate names like weight in this example. For survival, the beta intercept is Females (indices 1-6) and for capture probability it is time 2 (index 13).

Now that we created the PIMs and DM we can run the model but before we do that I want to show you the input file that the MARK interface is constructing. Press the Yellow arrow like icon at the top of the DM window and the following window will appear.

Setup Numerical Estimation Run

Title for Analyses

Model Name

Link Function

- ☐ Sin
- ☒ Logit
- ☐ LogLog
- ☐ CLogLog
- ☐ Log
- ☐ Identity
- ☐ Absolute
- ☐ Parm-Specific

Var. Estimation

- ☐ Hessian
- ☒ 2ndPart

☐ MCMC Estimation

Numerical Estimation Options:

- ☐ List Data
- ☐ Provide initial parameter estimates
- ☐ Use Alt. Opt. Method
- ☐ Profile Likelihood CI
- ☐ Set digits in estimates
- ☐ Set function evaluations
- ☐ Set number of parameters
- ☐ Standardize Individual Covariates
- ☐ Do not standardize design matrix

Real Par. Estimates from Individual Covariates

- ☐ First Encounter History Covariate Values
- ☒ Mean Individual Covariate Values
- ☐ User-specified Covariate Values

You'll want to add a title for the analysis and a model name. For the model name we will use `Phi(sex+weight)p(time+sex)` as a convention. Notice that under Link Function, MARK has selected Logit for the parameters based on the knowledge that we are using a DM for the model. Had we constructed a model solely with PIMS and used an identity DM, it would have selected the sin link but it is grayed out. There are various options on this page like Fixing real parameters, Variance estimation, Numerical estimation options, and Computation of real parameter values for individual covariates, but we will ignore those and press Ok to Save. When you do that the following Results Browser window will appear with our model highlighted in Red because it has only been saved and not run yet.

Model	AICc	Delta AICc	AICc Weight	Model Likelihood	No. Par.	Deviance
NO ESTIMATES (Phi(sex+weight)p(time+sex))	0.0000	*****	1.00000	1.0000	0	0.0000

Next press on the icon to the right of the trashcan icon and a notepad window will open as shown below.

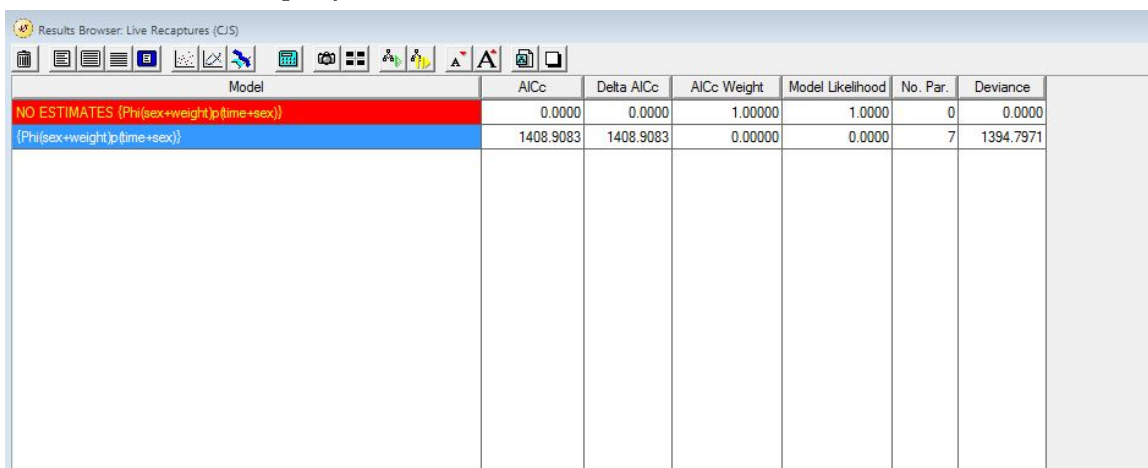
```

mrk5534z.tmp - Notepad
File Edit Format View Help
proc title Example analysis;
proc chmatrix occasions=4 groups=2 etype=Live mixtures=2 icovar=1 ICMeans NOhist hist=600;
  glabel(1)=Female;
  glabel(2)=Male;
  time interval 1 1 1;
  icovariates weight;
0011 1 0 2.96;
0011 1 0 3.09;
0011 1 0 3.14;
0011 1 0 3.23;
0010 1 0 3.47;
0010 1 0 3.5;
0011 1 0 3.6;
0011 1 0 3.61;
0011 1 0 3.65;
0011 1 0 3.68;
0011 1 0 3.7;
0011 1 0 3.72;
0010 1 0 3.79;
0010 1 0 3.84;
0011 1 0 3.89;
0010 1 0 3.96;
0011 1 0 3.98;
0011 1 0 4.01;
0011 1 0 4.06;
0011 1 0 4.09;
0011 1 0 4.09;
0011 1 0 4.11;
0010 1 0 4.13;
0010 1 0 4.2;
0010 1 0 4.2;
0011 1 0 4.21;
0010 1 0 4.25;
0011 1 0 4.31;
0011 1 0 4.31;
0011 1 0 4.32;
0011 1 0 4.33;
0010 1 0 4.34;
0010 1 0 4.36;
0010 1 0 4.46;
0010 1 0 4.46;

```

If you scroll down you'll see some header lines, the data, the PIMS, the design matrix and labels. This is the file that the interface created and when we run the model this file is passed to mark.exe which reads it in and fits the model. RMark is an interface that replaces everything we have done so far. It gathers all of the information from you about your data and model and constructs the input file for mark.exe and like the next steps, it runs the model and extracts the information from the output files that mark.exe creates.

So next, close the notepad window and press the green arrow icon to run the model. A similar Run window will appear and this time press Ok to Run. Depending on the speed of your computer, you may or may not see a black window box open and quickly close. That was mark.exe running the analysis. With longer running analysis you can monitor the progress of the analysis in that window. When it is completed, you'll see a window asking whether you want to append the model output to your results. Press Yes and you should see the following in your Results Browser window.



Model	AICc	Delta AICc	AICc Weight	Model Likelihood	No. Par.	Deviance
NO ESTIMATES (Phi(sex+weight)p(time+sex))	0.0000	0.0000	1.00000	1.0000	0	0.0000
{Phi(sex+weight)p(time+sex)}	1408.9083	1408.9083	0.00000	0.0000	7	1394.7971

You can delete the saved model (in red) by right clicking on it. With the blue model that was run, you can use the icons and menu items to examine the output or specifically parameter estimates, plot residuals, predict real parameter values based on covariates etc. If you run additional models they will appear in the results browser in order of increasing AICc which is the model selection criterion corrected for sample size. If you want to know more about how to use the MARK interface you can either go to one of the 5 day workshops that Gary White and others provide or read the 1000+ page book on MARK (<http://www.phidot.org/software/mark/docs/book/>). The book is a good resource about the various types of capture-recapture models and various analysis concepts even if you are using RMark. Most of the

example data sets that accompany the MARK book are provided either as examples with RMark (`?dipper`) or just the data sets (`?convert.inp`). You can learn and become more comfortable with RMark by comparing results in the book with your attempts to construct the same model in RMark.

5 Analyzing the Simple CJS Example with RMark

Everything we did with MARK needs to be accomplished with RMark but I think you'll find it much easier once you learn the R syntax. We need to provide the data, assign the data and model attributes, describe and run models and examine the output.

5.1 Providing the data

You can use any of the R functions like `read.table` or `read.delim` to read in the data. Alternatively you can use the functions `convert.inp` if you have an existing file in MARK format or `import.chdata` (a function in the RMark package) if the data are in RMark format. First I'll describe the RMark data format which is fairly simple for almost all of the models. I will not address nest survival data which is completely different but you can see `?killdeer` or `?mallard` for examples. For all of the remainder of the models, the only required field in the data is the capture history which must be named `ch` and it must be a character variable. It is important to realize that the default behavior of R is to read character data into dataframes as factor variables which are numeric variables with a character label. It is essential that the capture history be read in as a character variable. The function `import.chdata` forces the first field (`ch`) to be character. However, you can do the same by using the `colClasses` argument in `read.table` or `read.delim`. The only other field which has a fixed name is `freq` which is the frequency of each capture history. The `freq` field is not needed if each capture history represents a single individual (`freq=1`). Losses on capture (caught but not subsequently released) are represented by a negative `freq` value. The remainder of any fields in the data can have any name or type. If they are factor variables they can be used to define groups in the data (e.g., `sex`) or if they are numeric variables they can be used as individual covariates (e.g. `weight`). Below is code that shows how our simple example data can either be read in with `read.table` or `import.chdata`. Note that before we try to use any functions in RMark we must load the package from the library with `library(RMark)`.

```
library(RMark)

## Loading required package: snowfall
## Loading required package: snow
## This is RMark 2.1.13

# read in the data; header=TRUE means first row is column names;
# colClasses specify that the fields are of type character (ch),
# factor (sex) and numeric (weight)
RMarkData=read.table(file="example.txt",header=TRUE,
                     colClasses=c("character","factor","numeric"))
head(RMarkData)

##      ch      sex weight
## 1 0011 Female    2.96
## 2 0010 Female    3.09
## 3 0010 Female    3.14
## 4 0010 Female    3.23
## 5 0011 Female    3.47
## 6 0011 Female    3.50

# use import.chdata to read in the data; ch is first field and is character
# field.types f and n are for sex and weight fields
```

```
# Here I'll use the name df so typing will be easier later
df=import.chdata("example.txt",field.types=c("f","n"))
head(df)

##      ch      sex weight
## 1 0011 Female   2.96
## 2 0010 Female   3.09
## 3 0010 Female   3.14
## 4 0010 Female   3.23
## 5 0011 Female   3.47
## 6 0011 Female   3.50
```

You can use the `str` and `summary` R functions to make sure your data have been read in properly.

```
str(df)

## 'data.frame': 600 obs. of 3 variables:
## $ ch      : chr "0011" "0010" "0010" "0010" ...
## $ sex      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
## $ weight: num 2.96 3.09 3.14 3.23 3.47 3.5 3.6 3.61 3.65 3.68 ...

summary(df)

##      ch              sex      weight
## Length:600      Female:300   Min.    :1.700
## Class :character Male :300   1st Qu.:4.317
## Mode  :character           Median :5.020
##                                Mean   :5.008
##                                3rd Qu.:5.652
##                                Max.   :8.340
```

5.2 Describing data and model attributes

As with the first screen in the MARK interface, we need to specify the type of model for the data and some attributes for the data and model. This is accomplished with the RMark function `process.data` which uses the `data` and `model` arguments along with other optional arguments to create a list with the data and its attributes. To see all of the possible arguments, use `?process.data` to see the help file. For our example we only need to specify three arguments as shown below:

```
dp=process.data(df,model="CJS",groups="sex")
str(dp)

## List of 15
## $ data          : 'data.frame': 600 obs. of 4 variables:
## ..$ ch          : chr [1:600] "0011" "0010" "0010" "0010" ...
## ..$ sex         : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
## ..$ weight      : num [1:600] 2.96 3.09 3.14 3.23 3.47 3.5 3.6 3.61 3.65 3.68 ...
## ..$ group       : Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 1 ...
## $ model          : chr "CJS"
## $ mixtures       : num 1
## $ freq           : 'data.frame': 600 obs. of 2 variables:
## ..$ sexFemale: num [1:600] 1 1 1 1 1 1 1 1 1 1 ...
## ..$ sexMale  : num [1:600] 0 0 0 0 0 0 0 0 0 0 ...
## $ nocc           : num 4
```

```
## $ nocc.secondary : NULL
## $ time.intervals : num [1:3] 1 1 1
## $ begin.time     : num 1
## $ age.unit       : num 1
## $ initial.ages   : num [1:2] 0 0
## $ group.covariates:'data.frame': 2 obs. of 1 variable:
## ..$ sex: Factor w/ 2 levels "Female","Male": 1 2
## $ nstrata        : num 1
## $ strata.labels   : NULL
## $ counts         : NULL
## $ reverse        : logi FALSE
```

The list of possible values for model are given in MarkModels.pdf. Notice that we didn't need to specify the number of capture occasions, number of groups or group labels or the individual covariates and their names. That is done automatically by the code. If you had different time intervals or animals of different ages or a multistate model then there are other arguments that would be used to define those attributes.

The next step is to create design data which doesn't have an equivalent step in the MARK interface but it is essential to the RMark approach to defining a model from formulas. Design data are data that are attached to the parameters in the model and thus are specific to the type of model (e.g., CJS, Closed etc) and its attributes. The following creates the design data for our example:

```
ddl=make.design.data(dp)
str(ddl)

## List of 3
## $ Phi      :'data.frame': 12 obs. of 11 variables:
## ..$ par.index : int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
## ..$ model.index: num [1:12] 1 2 3 4 5 6 7 8 9 10 ...
## ..$ group      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## ..$ cohort     : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 3 1 1 1 2 ...
## ..$ age        : Factor w/ 3 levels "0","1","2": 1 2 3 1 2 1 1 2 3 1 ...
## ..$ time       : Factor w/ 3 levels "1","2","3": 1 2 3 2 3 3 1 2 3 2 ...
## ..$ occ.cohort : num [1:12] 1 1 1 2 2 3 1 1 1 2 ...
## ..$ Cohort     : num [1:12] 0 0 0 1 1 2 0 0 0 1 ...
## ..$ Age        : num [1:12] 0 1 2 0 1 0 0 1 2 0 ...
## ..$ Time       : num [1:12] 0 1 2 1 2 2 0 1 2 1 ...
## ..$ sex        : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## $ p          :'data.frame': 12 obs. of 11 variables:
## ..$ par.index : int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
## ..$ model.index: num [1:12] 13 14 15 16 17 18 19 20 21 22 ...
## ..$ group      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## ..$ cohort     : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 3 1 1 1 2 ...
## ..$ age        : Factor w/ 3 levels "1","2","3": 1 2 3 1 2 1 1 2 3 1 ...
## ..$ time       : Factor w/ 3 levels "2","3","4": 1 2 3 2 3 3 1 2 3 2 ...
## ..$ occ.cohort : num [1:12] 1 1 1 2 2 3 1 1 1 2 ...
## ..$ Cohort     : num [1:12] 0 0 0 1 1 2 0 0 0 1 ...
## ..$ Age        : num [1:12] 1 2 3 1 2 1 1 2 3 1 ...
## ..$ Time       : num [1:12] 0 1 2 1 2 2 0 1 2 1 ...
## ..$ sex        : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## $ pimtypes:List of 2
## ..$ Phi:List of 1
## .. ..$ pim.type: chr "all"
## ..$ p :List of 1
## .. ..$ pim.type: chr "all"
```

You can use any name but I tend to use `ddl` which stands for design data list because as shown above, the function returns a list with a dataframe for each parameter and then a list of `pimtypes`. The parameters for the CJS model are `Phi` and `p`, so the `ddl` contains a design dataframe `ddl$Phi` and `ddl$p`.

So what are these dataframes? The dataframes assign “data” to each real parameter in the model. The “data” that is assigned depends on the type of model (e.g., CJS vs Closed) as well as the parameter in the model. For the CJS model, we can define the following “data” for each parameter index:

1. the group which is either Female or Male
2. the release cohort occasion which is constant in a row of the PIM
3. the occasion (time) which is constant in a column of the PIM
4. the age or time since released which is constant along a diagonal of the PIM.

Each one of those fields is represented as a factor variable (`group`, `cohort`, `age`, `time`) and some as a numeric variable (`Cohort`, `Age`, `Time`) which enable fitting trend models. Using a capital letter is a convention from MARK where $p(t)$ means capture probability varies separately for each time and $p(T)$ is a trend with only an intercept and slope. The values of `cohort`, `age`, and `time` and their numeric counterparts depend on the value of `begin.time` and `time.intervals` specified for the model. However the variable `occ.cohort` is always the row number in the PIM. Note that the value of `time` and `age` differ for `Phi` and `p` because survival (`Phi`) refers to the survival in the interval between occasions and its value is for the `time` and `age` at the beginning of the interval; whereas, capture probability refers to a point in time. However, the values of the numeric equivalents like `Time` are the same for both `Phi` and `p` because `Time` is always given an origin of 0 which typically makes the intercept more useful. For example, if `begin.time=1990` and if `Time` started at 1990, the intercept for a trend model would be the value extrapolated back to the year 0 (1990 years previous) which would not be very useful. By using the origin 0 for `Time`, in any trend model the intercept is for `begin.time` for an interval parameter like `Phi` and `begin.time+time.intervals[1]` for `p`. Additionally it also avoids numerical issues when the value of `time` is large. For the same reasons, `Cohort` and `Age` are treated the same.

In this example, `group` and `sex` have the same values but if groups were defined based on more than one factor variable, then the `group` field would be the concatenated value of each combination (e.g., `MaleArea1`) and each factor variable would be included separately (e.g., `sex=MALE,Area=1`). These are the default data that are created specifically for the type of model specified in `process.data`. However, you can add any additional design data fields that you want and I’ll show examples of this later.

The `pimtype` is the default value of `all` which means all-different indices. You can set the `pimtype` to either `time` or `constant` but that restricts the type of models that can be constructed and those values are only used rarely to restrict the number of parameters.

Each row in the design dataframes has 2 indices. The `par.index` is the index for each parameter within that parameter type; whereas `model.index` is the unique index across all of the parameters in the model. The `model.index` values are the unique indices across all parameters in the model. They match the values in the PIMs that we defined earlier in MARK. For the first parameter the value of `par.index` and `model.index` are the same because the `model.index` values are assigned sequentially starting with the first parameter. If you see code in examples (particularly fixing real parameters), help files and older documentation for RMark that looks something like `as.numeric(row.names(ddl$...))`, that code is no longer needed because it was constructing the `par.index` value on the fly and it is now contained in the design data. In other functions within RMark you will see the label `all.diff.index` which is the same as `model.index`. Unfortunately I have not been consistent with the use of `par.index` so be aware that its meaning as a label can change. You will see another index labeled `vcv.index` which is the index in the variance-covariance matrix if one is provided from the function.

5.3 Describing and running a model

Running a simple model with default formulas is quite simple. We call the `mark` function with the arguments `dp` and `ddl` and assign it to an object with name `model` (or any name). A brief summary of the model is printed by default.

```

model=mark(dp,ddl)

##
## Output summary for CJS model
## Name : Phi(~1)p(~1)
##
## Npar : 2
## -2lnL: 1525.035
## AICc : 1529.047
##
## Beta
##           estimate      se      lcl      ucl
## Phi:(Intercept) 2.9193645 0.2651802 2.3996112 3.439118
## p:(Intercept)   0.9221261 0.0898523 0.7460156 1.098237
##
##
## Real Parameter Phi
## Group:sexFemale
##           1           2           3
## 1 0.9487954 0.9487954 0.9487954
## 2           0.9487954 0.9487954
## 3           0.9487954
##
## Group:sexMale
##           1           2           3
## 1 0.9487954 0.9487954 0.9487954
## 2           0.9487954 0.9487954
## 3           0.9487954
##
##
## Real Parameter p
## Group:sexFemale
##           2           3           4
## 1 0.7154751 0.7154751 0.7154751
## 2           0.7154751 0.7154751
## 3           0.7154751
##
## Group:sexMale
##           2           3           4
## 1 0.7154751 0.7154751 0.7154751
## 2           0.7154751 0.7154751
## 3           0.7154751

```

The default formulas for this model are constant survival and capture probability. The output shows the formulas used $\text{Phi}(\sim 1)p(\sim 1)$ where ~ 1 is the intercept (constant) model. Then it shows the number of parameters (2), the $-2 \times \log$ likelihood and AICc for model selection. That is followed by the estimates, standard errors and confidence intervals for the beta parameters (values on the logit scale) and finally by the real parameter values in triangular PIM format for Females and Males for Phi and p. This is a constant model so the values for Phi are the same for each sex and for all indices and likewise for p. You can compute the real parameter value for Phi directly from $\text{plogis}(2.9194) = 0.9488$.

So what did the `mark` function do? It constructed an input file for MARK using the same structure we saw earlier with the Save Structure in the MARK interface. The `mark` function then ran `mark.exe` and extracted the output from the files with extensions `.out` and `.vcv` and the result was stored in a list which was stored in the object `model`. The structure of the `model` object is shown below.


```

str(model)

## List of 25
## $ data :List of 15
## ..$ data :'data.frame': 600 obs. of 4 variables:
## .. ..$ ch : chr [1:600] "0011" "0010" "0010" "0010" ...
## .. ..$ sex : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
## .. ..$ weight: num [1:600] 2.96 3.09 3.14 3.23 3.47 3.5 3.6 3.61 3.65 3.68 ...
## .. ..$ group : Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 1 ...
## ..$ model : chr "CJS"
## ..$ mixtures : num 1
## ..$ freq : 'data.frame': 600 obs. of 2 variables:
## .. ..$ sexFemale: num [1:600] 1 1 1 1 1 1 1 1 1 1 ...
## .. ..$ sexMale : num [1:600] 0 0 0 0 0 0 0 0 0 0 ...
## ..$ nocc : num 4
## ..$ nocc.secondary : NULL
## ..$ time.intervals : num [1:3] 1 1 1
## ..$ begin.time : num 1
## ..$ age.unit : num 1
## ..$ initial.ages : num [1:2] 0 0
## ..$ group.covariates:'data.frame': 2 obs. of 1 variable:
## .. ..$ sex: Factor w/ 2 levels "Female","Male": 1 2
## ..$ nstrata : num 1
## ..$ strata.labels : NULL
## ..$ counts : NULL
## ..$ reverse : logi FALSE
## $ model : chr "CJS"
## $ title : chr ""
## $ model.name : chr "Phi(~1)p(~1)"
## $ links : chr "logit"
## $ mixtures : NULL
## $ call : language mark(data = dp, ddl = ddl)
## $ parameters :List of 2
## ..$ Phi:List of 7
## .. ..$ begin : num 0
## .. ..$ num : num -1
## .. ..$ default : num 1
## .. ..$ type : chr "Triang"
## .. ..$ pim.type: chr "all"
## .. ..$ link : chr "logit"
## .. ..$ formula :Class 'formula' length 2 ~1
## .. .. attr(*, ".Environment")=<environment: 0x07dbb5bc>
## ..$ p :List of 7
## .. ..$ begin : num 1
## .. ..$ num : num -1
## .. ..$ default : num 0
## .. ..$ type : chr "Triang"
## .. ..$ pim.type: chr "all"
## .. ..$ link : chr "logit"
## .. ..$ formula :Class 'formula' length 2 ~1
## .. .. attr(*, ".Environment")=<environment: 0x07dbb5bc>
## $ time.intervals : num [1:3] 1 1 1
## $ number.of.groups: int 2
## $ group.labels : chr [1:2] "sexFemale" "sexMale"

```

```

## $ nocc          : num 4
## $ begin.time    : num 1
## $ covariates    : NULL
## $ fixed         : NULL
## $ design.matrix : chr [1:2, 1:2] "1" "0" "0" "1"
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:2] "Phi gFemale c1 a0 t1" "p gFemale c1 a1 t2"
## .. ..$ : chr [1:2] "Phi:(Intercept)" "p:(Intercept)"
## $ pims          :List of 2
## ..$ Phi:List of 2
## .. ..$ :List of 2
## .. .. ..$ pim : num [1:3, 1:3] 1 0 0 2 4 0 3 5 6
## .. .. ..$ group: int 1
## .. .. ..$ :List of 2
## .. .. ..$ pim : num [1:3, 1:3] 7 0 0 8 10 0 9 11 12
## .. .. ..$ group: int 2
## ..$ p :List of 2
## .. ..$ :List of 2
## .. .. ..$ pim : num [1:3, 1:3] 13 0 0 14 16 0 15 17 18
## .. .. ..$ group: int 1
## .. .. ..$ :List of 2
## .. .. ..$ pim : num [1:3, 1:3] 19 0 0 20 22 0 21 23 24
## .. .. ..$ group: int 2
## $ design.data   :List of 3
## ..$ Phi         : 'data.frame': 12 obs. of  11 variables:
## .. ..$ par.index : int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
## .. ..$ model.index: num [1:12] 1 2 3 4 5 6 7 8 9 10 ...
## .. ..$ group      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## .. ..$ cohort     : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 3 1 1 1 2 ...
## .. ..$ age        : Factor w/ 3 levels "0","1","2": 1 2 3 1 2 1 1 2 3 1 ...
## .. ..$ time       : Factor w/ 3 levels "1","2","3": 1 2 3 2 3 3 1 2 3 2 ...
## .. ..$ occ.cohort : num [1:12] 1 1 1 2 2 3 1 1 1 2 ...
## .. ..$ Cohort     : num [1:12] 0 0 0 1 1 2 0 0 0 1 ...
## .. ..$ Age        : num [1:12] 0 1 2 0 1 0 0 1 2 0 ...
## .. ..$ Time       : num [1:12] 0 1 2 1 2 2 0 1 2 1 ...
## .. ..$ sex        : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## ..$ p            : 'data.frame': 12 obs. of  11 variables:
## .. ..$ par.index : int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
## .. ..$ model.index: num [1:12] 13 14 15 16 17 18 19 20 21 22 ...
## .. ..$ group      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## .. ..$ cohort     : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 3 1 1 1 2 ...
## .. ..$ age        : Factor w/ 3 levels "1","2","3": 1 2 3 1 2 1 1 2 3 1 ...
## .. ..$ time       : Factor w/ 3 levels "2","3","4": 1 2 3 2 3 3 1 2 3 2 ...
## .. ..$ occ.cohort : num [1:12] 1 1 1 2 2 3 1 1 1 2 ...
## .. ..$ Cohort     : num [1:12] 0 0 0 1 1 2 0 0 0 1 ...
## .. ..$ Age        : num [1:12] 1 2 3 1 2 1 1 2 3 1 ...
## .. ..$ Time       : num [1:12] 0 1 2 1 2 2 0 1 2 1 ...
## .. ..$ sex        : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## ..$ pimtypes:List of 2
## .. ..$ Phi:List of 1
## .. .. ..$ pim.type: chr "all"
## .. ..$ p :List of 1
## .. .. ..$ pim.type: chr "all"

```

```
## $ strata.labels : NULL
## $ mlogit.list :List of 2
## ..$ structure: NULL
## ..$ ncol : num 1
## $ profile.int : logi FALSE
## $ simplify :List of 2
## ..$ pim.translation: int [1:24] 1 1 1 1 1 1 1 1 1 1 ...
## ..$ real.labels : chr [1:24] "Phi gFemale c1 a0 t1" "Phi gFemale c1 a1 t2" "Phi gFemale c1 a2 t3"
## $ model.parameters: list()
## $ results :List of 14
## ..$ lnL : num 1525
## ..$ deviance : num 153
## ..$ deviance.df : num 20
## ..$ npar : num 2
## ..$ n : int 1012
## ..$ AICc : num 1529
## ..$ beta : 'data.frame': 2 obs. of 4 variables:
## .. ..$ estimate: num [1:2] 2.919 0.922
## .. ..$ se : num [1:2] 0.2652 0.0899
## .. ..$ lcl : num [1:2] 2.4 0.746
## .. ..$ ucl : num [1:2] 3.44 1.1
## ..$ real : 'data.frame': 2 obs. of 6 variables:
## .. ..$ estimate: num [1:2] 0.949 0.715
## .. ..$ se : num [1:2] 0.0129 0.0183
## .. ..$ lcl : num [1:2] 0.917 0.678
## .. ..$ ucl : num [1:2] 0.969 0.75
## .. ..$ fixed : Factor w/ 1 level " ": 1 1
## .. ..$ note : Factor w/ 1 level " ": 1 1
## ..$ beta.vcv : num [1:2, 1:2] 0.07032 -0.0134 -0.0134 0.00807
## ..$ derived : NULL
## ..$ derived.vcv : NULL
## ..$ covariate.values: NULL
## ..$ singular : NULL
## ..$ real.vcv : NULL
## $ output : chr "mark004"
## - attr(*, "class")= chr [1:2] "mark" "CJS"
```

The model object is self-contained with all of the data and attributes including:

- **data:** the processed data and its attributes
- **parameters:** which is the structure of the parameters used in the model
- **design.matrix:** the design matrix for the model
- **design.data:** the design data used to construct the model
- **pims:** the model PIMs
- **results:** a list containing all of the results from MARK including the beta and real estimates including standard errors and confidence intervals and the variance-covariance matrix.
- **output:** the base name of the input and output files (in this case it is mark004 with extensions `.inp`, `.out`, `.vcv`, `.res`); these files are kept in the working directory but are typically only needed if there was an error in constructing the model. Viewing the output file can actually be confusing which I'll explain in section 6.10. However, you can easily view the output by typing `model` (or whatever you

named the result) and the output will be opened in a notepad window. If an error occurred then model would not have been constructed and you'll have to open the *.out file directly.

5.4 Examining the model output

You can extract the estimates directly from the results, but it is best to use the RMark functions `coef` and `summary` to extract the parameter estimates. The reason is described in section 10. The `coef` function extracts the beta estimates:

```
coef(model)

##              estimate          se          lcl          ucl
## Phi:(Intercept) 2.9193645 0.2651802 2.3996112 3.439118
## p:(Intercept)   0.9221261 0.0898523 0.7460156 1.098237
```

The `summary` function produces a list with various results. By default the summary results are printed as with any R object:

```
summary(model)

## Output summary for CJS model
## Name : Phi(~1)p(~1)
##
## Npar : 2
## -2lnL: 1525.035
## AICc : 1529.047
##
## Beta
##              estimate          se          lcl          ucl
## Phi:(Intercept) 2.9193645 0.2651802 2.3996112 3.439118
## p:(Intercept)   0.9221261 0.0898523 0.7460156 1.098237
##
##
## Real Parameter Phi
## Group:sexFemale
##           1           2           3
## 1 0.9487954 0.9487954 0.9487954
## 2           0.9487954 0.9487954
## 3           0.9487954
##
## Group:sexMale
##           1           2           3
## 1 0.9487954 0.9487954 0.9487954
## 2           0.9487954 0.9487954
## 3           0.9487954
##
##
## Real Parameter p
## Group:sexFemale
##           2           3           4
## 1 0.7154751 0.7154751 0.7154751
## 2           0.7154751 0.7154751
## 3           0.7154751
##
```

```
## Group:sexMale
##           2           3           4
## 1 0.7154751 0.7154751 0.7154751
## 2           0.7154751 0.7154751
## 3           0.7154751
```

However, you can extract any of the **summary** function components which are shown with the **str** function:

```
str(summary(model))

## List of 10
## $ model      : chr "CJS"
## $ title      : chr ""
## $ model.name: chr "Phi(~1)p(~1)"
## $ model.call: language mark(data = dp, ddl = ddl)
## $ npar       : num 2
## $ ln1        : num 1525
## $ AICc       : num 1529
## $ beta       : 'data.frame': 2 obs. of  4 variables:
## ..$ estimate: num [1:2] 2.919 0.922
## ..$ se       : num [1:2] 0.2652 0.0899
## ..$ lcl      : num [1:2] 2.4 0.746
## ..$ ucl      : num [1:2] 3.44 1.1
## $ reals      :List of 2
## ..$ Phi:List of 2
## .. ..$ Group:sexFemale:List of 2
## .. .. ..$ pim : num [1:3, 1:3] 0.949 NA NA 0.949 0.949 ...
## .. .. ..- attr(*, "dimnames")=List of 2
## .. .. .. ..$ : chr [1:3] "1" "2" "3"
## .. .. .. ..$ : chr [1:3] "1" "2" "3"
## .. .. ..$ group: int 1
## .. ..$ Group:sexMale :List of 2
## .. .. ..$ pim : num [1:3, 1:3] 0.949 NA NA 0.949 0.949 ...
## .. .. ..- attr(*, "dimnames")=List of 2
## .. .. .. ..$ : chr [1:3] "1" "2" "3"
## .. .. .. ..$ : chr [1:3] "1" "2" "3"
## .. .. ..$ group: int 2
## ..$ p :List of 2
## .. ..$ Group:sexFemale:List of 2
## .. .. ..$ pim : num [1:3, 1:3] 0.715 NA NA 0.715 0.715 ...
## .. .. ..- attr(*, "dimnames")=List of 2
## .. .. .. ..$ : chr [1:3] "1" "2" "3"
## .. .. .. ..$ : chr [1:3] "2" "3" "4"
## .. .. ..$ group: int 1
## .. ..$ Group:sexMale :List of 2
## .. .. ..$ pim : num [1:3, 1:3] 0.715 NA NA 0.715 0.715 ...
## .. .. ..- attr(*, "dimnames")=List of 2
## .. .. .. ..$ : chr [1:3] "1" "2" "3"
## .. .. .. ..$ : chr [1:3] "2" "3" "4"
## .. .. ..$ group: int 2
## $ brief      : logi FALSE
## - attr(*, "class")= chr "summary.mark"

summary(model)$AICc
## [1] 1529.047
```

6 Design Data and Formulas

So far we have only fit a very simple default model. Now we will discuss how you can fit just about any model that you can imagine. You can certainly do the same with the MARK interface but here we introduce the use of formulas so you can avoid creating DMs manually. By this point you should understand the relationship between PIMs and DMs. If not, go re-read the previous sections. I'll start by describing the R function `model.matrix` which is an integral component of RMark.

6.1 What is `model.matrix`?

The R function `model.matrix` creates a design matrix from a formula applied to data. You do not need to use `model.matrix` to create a model with RMark because it does that for you. However, unless you are using individual covariates, you can try out your formula with `model.matrix` to help you understand the DM that your formula will create. I will use `model.matrix` here to explain the link between formulas and design matrices, so you have a better understanding of what RMark is doing. Many other modeling packages also use `model.matrix` so it will help your understanding in general with specifying models and interpreting the coefficients. I have found that folks with only one or two statistics courses generally lack this understanding and MARK is often their first encounter with DMs. Even though RMark creates the DM for you, you need to understand what you are doing.

Before jumping into use of `model.matrix` with RMark, I'll give a simple example to describe the important difference between factor and numeric variables. The focus here is the independent predictor variables and not the dependent variable. For more details see `LinearModels.pdf` in the RMark documentation archive. Consider the following simple set of data:

```
data=expand.grid(sex=c("Female","Male"),age=c("0","1","2","3"))
set.seed(98215)
data$height=rnorm(8,10,3)
data

##      sex age   height
## 1 Female  0 12.273859
## 2 Male   0  9.150697
## 3 Female  1  8.179441
## 4 Male   1  8.141125
## 5 Female  2  8.645582
## 6 Male   2 16.343100
## 7 Female  3 10.784280
## 8 Male   3 10.101928

str(data)

## 'data.frame': 8 obs. of  3 variables:
## $ sex : Factor w/ 2 levels "Female","Male": 1 2 1 2 1 2 1 2
## $ age : Factor w/ 4 levels "0","1","2","3": 1 1 2 2 3 3 4 4
## $ height: num 12.27 9.15 8.18 8.14 8.65 ...
## - attr(*, "out.attrs")=List of 2
## ..$ dim : Named int 2 4
## ..$ attr(*, "names")= chr "sex" "age"
## ..$ dimnames:List of 2
## ..$ sex: chr "sex=Female" "sex=Male"
## ..$ age: chr "age=0" "age=1" "age=2" "age=3"
```

Notice that `sex` and `age` are factor variables and `height` is a numeric variable. The variable `sex` has 2 levels and the variable `age` has 4 levels. Factor variables define a subset of the data and when a factor with k levels is used in a formula with `model.matrix` and treatment contrast (see `?options` and `?contrasts` in R) $k - 1$

dummy variables are created in addition to the intercept. Thus if you use the formulas `~sex` and `~age` with `model.matrix` with data they create design matrices with 2 (intercept + 1) and 4 (intercept +3) columns respectively:

```
model.matrix(~sex,data)

##      (Intercept) sexMale
## 1             1      0
## 2             1      1
## 3             1      0
## 4             1      1
## 5             1      0
## 6             1      1
## 7             1      0
## 8             1      1
## attr("assign")
## [1] 0 1
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"

model.matrix(~age,data)

##      (Intercept) age1 age2 age3
## 1             1     0     0     0
## 2             1     0     0     0
## 3             1     1     0     0
## 4             1     1     0     0
## 5             1     0     1     0
## 6             1     0     1     0
## 7             1     0     0     1
## 8             1     0     0     1
## attr("assign")
## [1] 0 1 1 1
## attr("contrasts")
## attr("contrasts")$age
## [1] "contr.treatment"
```

Notice that the intercept represents the first level of the factor variable which is Female for `sex` and 0 for `age`. If you remove the intercept with a factor variable `model.matrix` still creates 2 columns in the DM but there is now a separate column for each factor level rather than specifying k-1 levels relative to an intercept:

```
model.matrix(~-1+sex,data)

##      sexFemale sexMale
## 1             1      0
## 2             0      1
## 3             1      0
## 4             0      1
## 5             1      0
## 6             0      1
## 7             1      0
## 8             0      1
## attr("assign")
## [1] 1 1
```

```
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"

model.matrix(~-1+age,data)

##   age0 age1 age2 age3
## 1    1    0    0    0
## 2    1    0    0    0
## 3    0    1    0    0
## 4    0    1    0    0
## 5    0    0    1    0
## 6    0    0    1    0
## 7    0    0    0    1
## 8    0    0    0    1
## attr("assign")
## [1] 1 1 1 1
## attr("contrasts")
## attr("contrasts")$age
## [1] "contr.treatment"
```

Now when you specify an additive formula with 2 factor variables, each factor provides $k-1$ dummy variables plus the intercept, so `~sex+age` creates 5 (1+1+3) columns:

```
model.matrix(~sex+age,data)

##   (Intercept) sexMale age1 age2 age3
## 1           1      0    0    0    0
## 2           1      1    0    0    0
## 3           1      0    1    0    0
## 4           1      1    1    0    0
## 5           1      0    0    1    0
## 6           1      1    0    1    0
## 7           1      0    0    0    1
## 8           1      1    0    0    1
## attr("assign")
## [1] 0 1 2 2 2
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"
##
## attr("contrasts")$age
## [1] "contr.treatment"
```

If you were to remove the intercept in a model with additive factor variables, the first factor variable has separate columns but they represent the first level of the second factor variable as shown below:

```
model.matrix(~-1+sex+age,data)

##   sexFemale sexMale age1 age2 age3
## 1          1      0    0    0    0
## 2          0      1    0    0    0
## 3          1      0    1    0    0
## 4          0      1    1    0    0
## 5          1      0    0    1    0
```



```
## 6      0      1      0      1      0
## 7      1      0      0      0      1
## 8      0      1      0      0      1
## attr("assign")
## [1] 1 1 2 2 2
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"
##
## attr("contrasts")$age
## [1] "contr.treatment"
```

The interaction model of 2 factor variables where they have k_1 and k_2 levels, you get $k_1 * k_2$ columns in the DM. You have an intercept (1), a main effect for each factor ($k_1 - 1 + k_2 - 1$), and the interactions ($(k_1 - 1) * (k_2 - 1)$). If you add the 3 values you'll see it equals $k_1 * k_2$ and an example of `~sex*age` is shown below:

```
model.matrix(~sex*age,data)

##      (Intercept) sexMale age1 age2 age3 sexMale:age1 sexMale:age2 sexMale:age3
## 1              1      0      0      0      0              0              0              0
## 2              1      1      0      0      0              0              0              0
## 3              1      0      1      0      0              0              0              0
## 4              1      1      1      0      0              1              0              0
## 5              1      0      0      1      0              0              0              0
## 6              1      1      0      1      0              0              1              0
## 7              1      0      0      0      1              0              0              0
## 8              1      1      0      0      1              0              0              1
## attr("assign")
## [1] 0 1 2 2 2 3 3 3
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"
##
## attr("contrasts")$age
## [1] "contr.treatment"
```

Now a subtlety different way to express the above model is `~-1+sex:age`. If you use `factor1:factor2` in a formula rather than `factor1*factor2`, `model.matrix` will create an intercept and $k_1 * k_2$ separate dummy variables that represent the subset of the data defined by both factor variables. The -1 is needed to remove the intercept which is necessary or you will have too many columns in the DM :

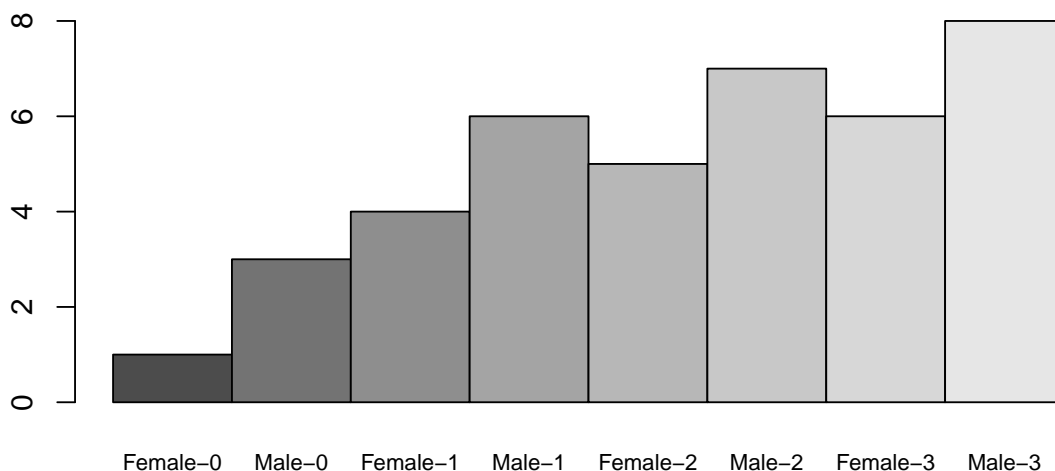
```
model.matrix(~-1+sex:age,data)

##      sexFemale:age0 sexMale:age0 sexFemale:age1 sexMale:age1 sexFemale:age2
## 1              1      0      0      0      0
## 2              0      1      0      0      0
## 3              0      0      1      0      0
## 4              0      0      0      1      0
## 5              0      0      0      0      1
## 6              0      0      0      0      0
## 7              0      0      0      0      0
## 8              0      0      0      0      0
##      sexMale:age2 sexFemale:age3 sexMale:age3
## 1              0      0      0
```

```
## 2      0      0      0
## 3      0      0      0
## 4      0      0      0
## 5      0      0      0
## 6      1      0      0
## 7      0      1      0
## 8      0      0      1
## attr("assign")
## [1] 1 1 1 1 1 1 1 1
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"
##
## attr("contrasts")$age
## [1] "contr.treatment"
```

Factor variables define a different value for each subset of the data defined by the levels of the factor variable(s). For example, if we consider the $\sim \text{sex} + \text{age}$ model with the parameter (β) values 1:5 then if X is the DM then $X\beta$ and a plot of the values is shown below:

```
X=model.matrix(~sex+age,data)
beta=1:5
Xbeta=X%*%beta
barplot(Xbeta,beside=TRUE,names.arg=paste(data$sex,data$age,sep="-"),cex.names=.75)
```

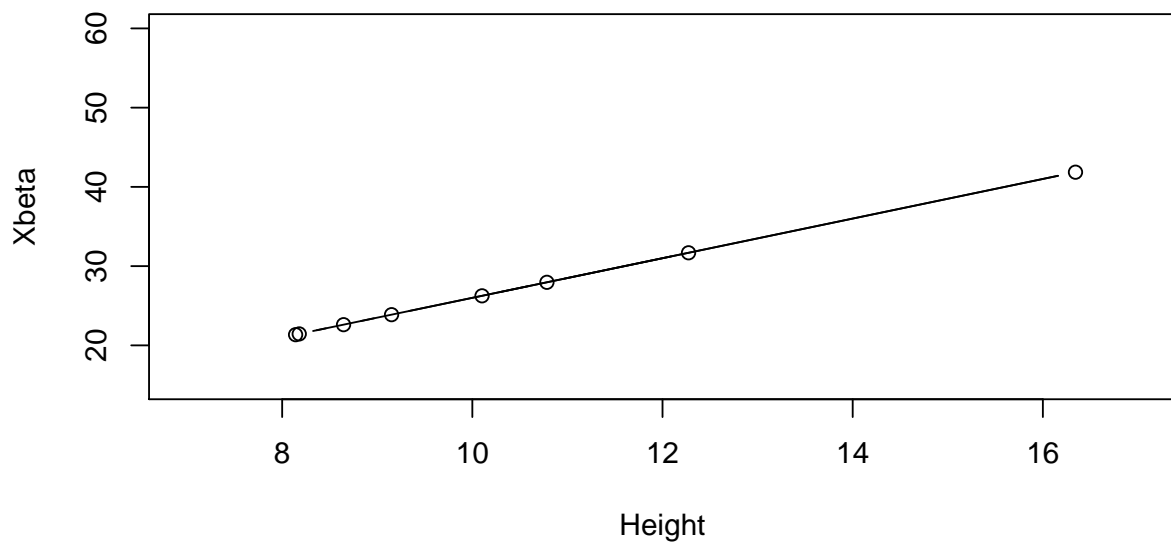


Now let's consider the numeric variable `height` in a formula. The beta value for a numeric variable is a slope whereas the factor levels adjusted the intercept:

```
X=model.matrix(~height,data)
X
##      (Intercept)      height
```

```
## 1      1 12.273859
## 2      1  9.150697
## 3      1  8.179441
## 4      1  8.141125
## 5      1  8.645582
## 6      1 16.343100
## 7      1 10.784280
## 8      1 10.101928
## attr("assign")
## [1] 0 1

beta=c(1,2.5)
Xbeta=X%*%beta
plot(data$height,Xbeta,xlab="Height",type="b",xlim=c(7,17),ylim=c(15,60))
```

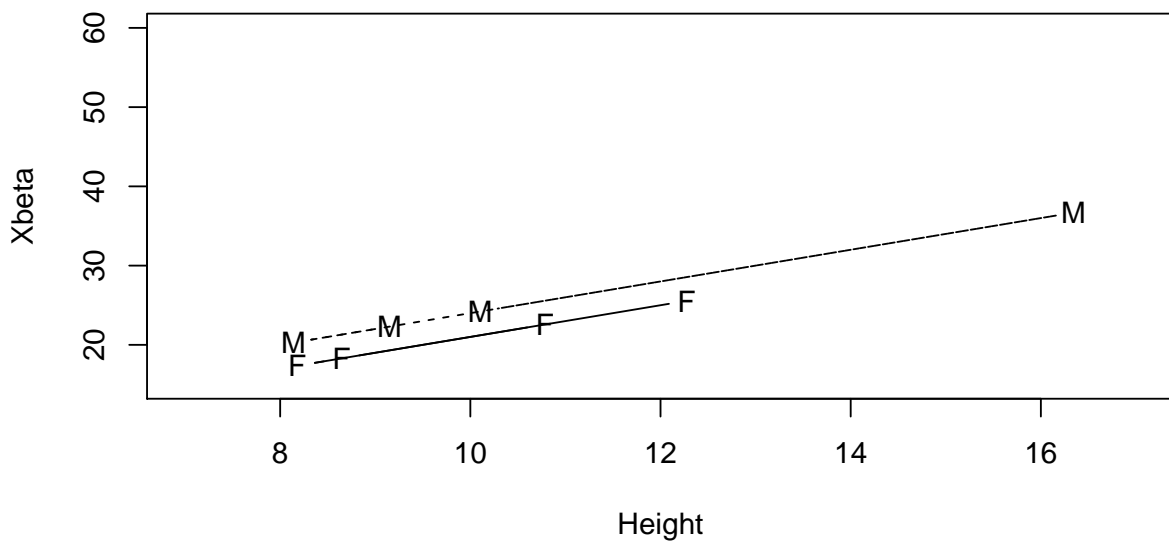


When we consider an additive model with a factor variable that has k levels and a numeric variable we get k lines that can have different intercepts:

```
X=model.matrix(~sex+height,data)
X
##      (Intercept) sexMale    height
## 1             1      0 12.273859
## 2             1      1  9.150697
## 3             1      0  8.179441
## 4             1      1  8.141125
## 5             1      0  8.645582
## 6             1      1 16.343100
## 7             1      0 10.784280
## 8             1      1 10.101928
## attr("assign")
## [1] 0 1 2
```

```
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"

beta=c(1,3,2)
Xbeta=X%%beta
plot(data$height[data$sex=="Female"],Xbeta[data$sex=="Female"],
      xlab="Height",type="b",pch="F",ylab="Xbeta",xlim=c(7,17),ylim=c(15,60))
lines(data$height[data$sex=="Male"],Xbeta[data$sex=="Male"],xlab="Height",
      type="b",pch="M",lty=2)
```



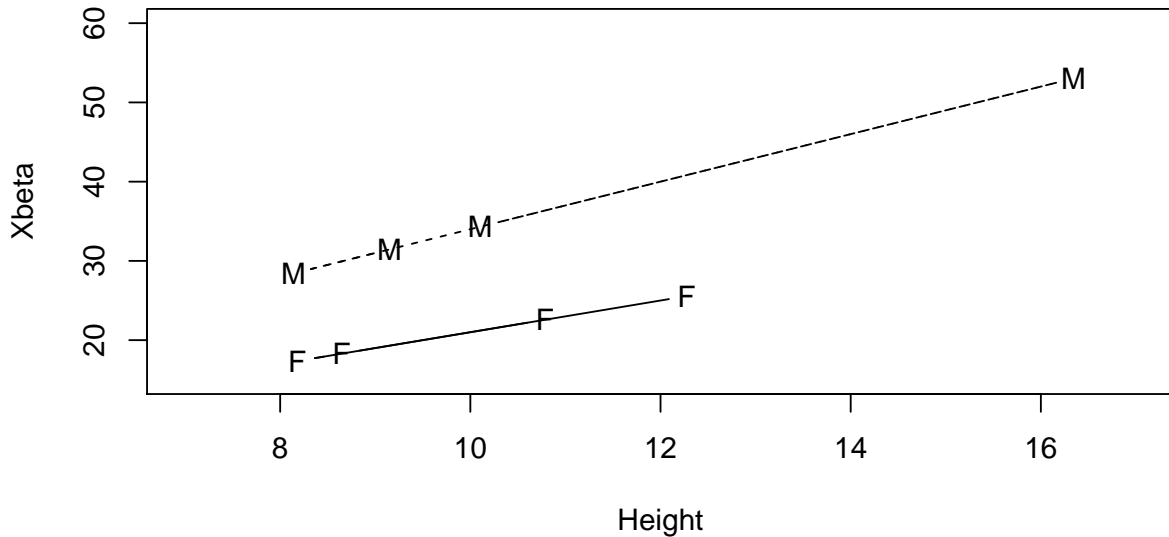
If you interact a factor and numeric variable in a formula you are specifying that the slope of the numeric variable can vary across levels of the factor variable as well:

```
X=model.matrix(~sex*height,data)
X
##      (Intercept) sexMale      height sexMale:height
## 1             1         0 12.273859         0.000000
## 2             1         1  9.150697         9.150697
## 3             1         0  8.179441         0.000000
## 4             1         1  8.141125         8.141125
## 5             1         0  8.645582         0.000000
## 6             1         1 16.343100        16.343100
## 7             1         0 10.784280         0.000000
## 8             1         1 10.101928        10.101928
## attr("assign")
## [1] 0 1 2 3
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"
```

```

beta=c(1,3,2,1)
Xbeta=X%*%beta
plot(data$height[data$sex=="Female"],Xbeta[data$sex=="Female"],
      xlab="Height",type="b",pch="F",ylab="Xbeta",xlim=c(7,17),ylim=c(15,60))
lines(data$height[data$sex=="Male"],Xbeta[data$sex=="Male"],xlab="Height",
      type="b",pch="M",lty=2)

```



If you specify the model as `~sex:height`, the lines can have different slopes but the same intercept:

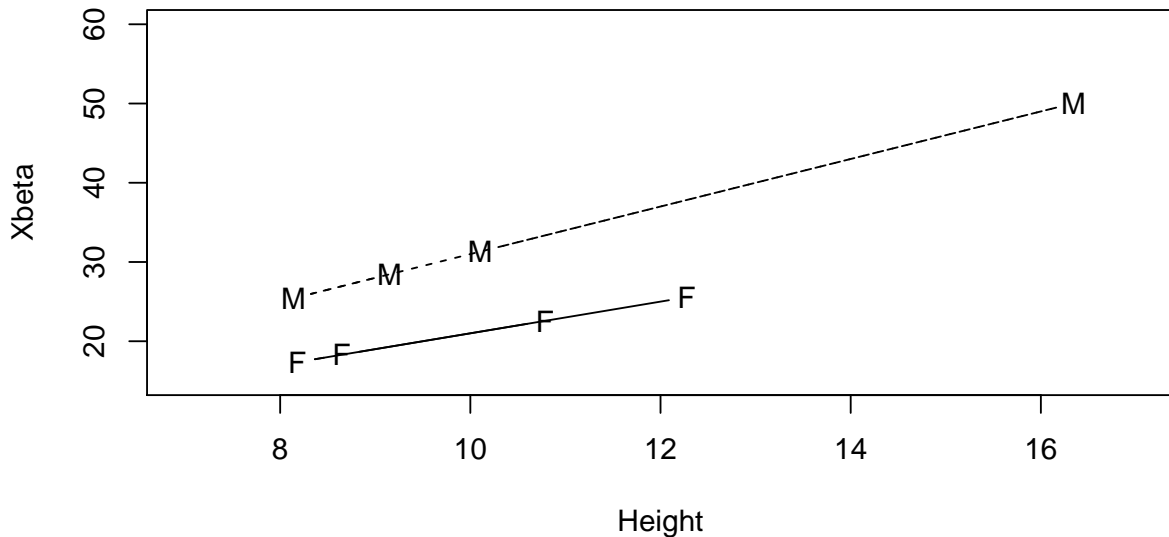
```

X=model.matrix(~sex:height,data)
X
##      (Intercept) sexFemale:height sexMale:height
## 1             1      12.273859      0.000000
## 2             1       0.000000      9.150697
## 3             1       8.179441      0.000000
## 4             1       0.000000      8.141125
## 5             1       8.645582      0.000000
## 6             1       0.000000     16.343100
## 7             1     10.784280      0.000000
## 8             1       0.000000     10.101928
## attr("assign")
## [1] 0 1 1
## attr("contrasts")
## attr("contrasts")$sex
## [1] "contr.treatment"

beta=c(1,2,3)
Xbeta=X%*%beta
plot(data$height[data$sex=="Female"],Xbeta[data$sex=="Female"],
      xlab="Height",type="b",pch="F",ylab="Xbeta",xlim=c(7,17),ylim=c(15,60))
lines(data$height[data$sex=="Male"],Xbeta[data$sex=="Male"],xlab="Height",

```

```
type="b",pch="M",lty=2)
```



Notice that the height slope columns are now separate rather than slope for females as an intercept with a difference added for the height slope for males, so the third beta for this model is $2+1$. If you were to plot for a range of heights starting with height=0, you would find that the lines would meet at height=0 because they have the same intercept.

For RMark models, the data are the design data and the individual covariates (e.g., **weight**). The latter cannot be handled directly by **model.matrix** without some trickery described later. For now we will focus on creating models from the design data. Our formulas can use any of the data in the design data and for the time being I'll focus on the default design data created by **make.design.data**. As a simple example, let's say that I want to make capture probability sex-specific. This can be accomplished with **model.matrix** as follows:

```
model.matrix(~sex,ddl$p)

##      (Intercept) sexMale
## 1             1      0
## 2             1      0
## 3             1      0
## 4             1      0
## 5             1      0
## 6             1      0
## 7             1      1
## 8             1      1
## 9             1      1
## 10            1      1
## 11            1      1
## 12            1      1
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
```

```
## attr("contrasts")$sex
## [1] "contr.treatment"
```

The function created a DM for p with a treatment contrast in which the intercept represents females and the sex column is the difference between males (last 6 rows) and females (first 6 rows). To solidify your understanding of the link between design data and the DM and to show how you can actually interject columns in the DM, I'll construct the same model for p in different ways with the following code:

```
# add the fields intercept and male to p design data
ddl$p$intercept=1
ddl$p$male=ifelse(ddl$p$sex=="Male",1,0)
# create the DM using an explicit dummy variable for males
model.matrix(~male,ddl$p)

##      (Intercept) male
## 1             1    0
## 2             1    0
## 3             1    0
## 4             1    0
## 5             1    0
## 6             1    0
## 7             1    1
## 8             1    1
## 9             1    1
## 10            1    1
## 11            1    1
## 12            1    1
## attr("assign")
## [1] 0 1

# create the DM using an explicit dummy variable for the intercept and males
model.matrix(~-1+intercept+male,ddl$p)

##      intercept male
## 1             1    0
## 2             1    0
## 3             1    0
## 4             1    0
## 5             1    0
## 6             1    0
## 7             1    1
## 8             1    1
## 9             1    1
## 10            1    1
## 11            1    1
## 12            1    1
## attr("assign")
## [1] 1 2
```

The -1 in the formula removed the intercept and it was explicitly entered with the dummy variable which had the value 1 for all p parameters in the design data. That is not particularly useful but having a dummy variable like **male** can be useful in cases where you want to restrict an interaction. Hopefully what is obvious to you now is that you can construct models from formulas using any of the fields in the design data and the **key** to creating most models in RMark is to create or modify the design data to meet your needs (section 6.4).

We could also create a sex-specific model for survival and create a complete DM for the 24 real parameters in the model with:

```
pdm=model.matrix(~sex,ddl$p)
Phidm=model.matrix(~sex,ddl$Phi)
dm=cbind(Phidm,matrix(0,nrow=12,ncol=2))
dm=rbind(dm,cbind(matrix(0,nrow=12,ncol=2),pdm))
colnames(dm)=c(paste("Phi:",colnames(Phidm),sep=""),paste("p:",colnames(pdm),sep=""))
rownames(dm)=1:nrow(dm)
dm
```

	Phi:(Intercept)	Phi:sexMale	p:(Intercept)	p:sexMale
## 1	1	0	0	0
## 2	1	0	0	0
## 3	1	0	0	0
## 4	1	0	0	0
## 5	1	0	0	0
## 6	1	0	0	0
## 7	1	1	0	0
## 8	1	1	0	0
## 9	1	1	0	0
## 10	1	1	0	0
## 11	1	1	0	0
## 12	1	1	0	0
## 13	0	0	1	0
## 14	0	0	1	0
## 15	0	0	1	0
## 16	0	0	1	0
## 17	0	0	1	0
## 18	0	0	1	0
## 19	0	0	1	1
## 20	0	0	1	1
## 21	0	0	1	1
## 22	0	0	1	1
## 23	0	0	1	1
## 24	0	0	1	1

What is done in the code above is similar to what RMark does in the function `make.mark.model` which is called from the `mark` function to create the model input for MARK. See section C.4 in Appendix C of the Cooch and White book for a more detailed discussion of what the `mark` function does.

6.2 Specifying RMark models with formulas

So, let's learn how to specify models with formulas for RMark. I'll use a specific naming convention that will be used later on in section 8. To create the above model with sex-specific survival and capture probability we create 2 lists with each containing a single element named `formula` and in this case the lists are identical but are stored in separate objects:

```
Phi.sex=list(formula=~sex)
p.sex=list(formula=~sex)
```

Now we will use those values to create and run the model with our example data using the `mark` function by specifying the argument `model.parameters` which is a list containing each parameter list (a list of lists):


```
sex.model=mark(dp,ddl,model.parameters=list(Phi=Phi.sex,p=p.sex))

##
## Output summary for CJS model
## Name : Phi(~sex)p(~sex)
##
## Npar : 4
## -2lnL: 1515.939
## AICc : 1523.979
##
## Beta
##           estimate      se      lcl      ucl
## Phi:(Intercept) 2.4135006 0.2895129 1.8460553 2.9809460
## Phi:sexMale      1.2704264 0.6632554 -0.0295542 2.5704069
## p:(Intercept)    0.8914318 0.1361638 0.6245508 1.1583129
## p:sexMale        0.0721623 0.1809107 -0.2824227 0.4267472
##
##
## Real Parameter Phi
## Group:sexFemale
##           1           2           3
## 1 0.917851 0.917851 0.917851
## 2           0.917851 0.917851
## 3           0.917851
##
## Group:sexMale
##           1           2           3
## 1 0.9754916 0.9754916 0.9754916
## 2           0.9754916 0.9754916
## 3           0.9754916
##
##
## Real Parameter p
## Group:sexFemale
##           2           3           4
## 1 0.7091856 0.7091856 0.7091856
## 2           0.7091856 0.7091856
## 3           0.7091856
##
## Group:sexMale
##           2           3           4
## 1 0.7238408 0.7238408 0.7238408
## 2           0.7238408 0.7238408
## 3           0.7238408

# Note in this case we could have done the same thing with:
# sex=list(formula=~sex)
# sex.model=mark(dp,ddl,model.parameters=list(Phi=sex,p=sex))
# But usually the models will differ and the reason for different
# specifications will become clear later.
```

Now we have 4 beta parameters and the real parameters in PIM format show the differences for males and females.

6.3 Individual covariates

Initially I handled individual covariates by adding columns in the DM outside of the formula because `model.matrix` cannot handle the MARK requirement to plug in the name of the individual covariate. That was a kludge and not very general so I devised a way to trick `model.matrix` by creating a temporary design data variable with the name of the individual covariate and a value of all 1's. I use `model.matrix` with the formula and design data including the temporary variable and then fill in the covariate name in the appropriate columns of the DM that `model.matrix` created. So let's see how that works by creating the model from section 4 with survival varying by sex and initial weight and capture probability varying by time with an additive sex effect. We start by defining the formulas for Phi and p:

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

Now because we are using the individual covariate weight, I cannot show you the DM created by `model.matrix` but we will use those formulas in the call to the mark function with our processed data and design data.

```
example.model=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.weight,p=p.sex.time))

##
## Output summary for CJS model
## Name : Phi(~sex + weight)p(~sex + time)
##
## Npar : 7
## -2lnL: 1380.878
## AICc : 1394.989
##
## Beta
##
## estimate se lcl ucl
## Phi:(Intercept) -0.1702522 0.7841879 -1.7072605 1.3667561
## Phi:sexMale 0.9565767 0.4108460 0.1513185 1.7618348
## Phi:weight 0.4971907 0.1709804 0.1620692 0.8323122
## p:(Intercept) -0.2353620 0.1780441 -0.5843284 0.1136044
## p:sexMale 0.0895407 0.1836570 -0.2704270 0.4495085
## p:time3 2.5057779 0.2613210 1.9935887 3.0179670
## p:time4 1.1106124 0.1899715 0.7382683 1.4829564
##
##
## Real Parameter Phi
## Group:sexFemale
## 1 2 3
## 1 0.9105093 0.9105093 0.9105093
## 2 0.9105093 0.9105093
## 3 0.9105093
##
## Group:sexMale
## 1 2 3
## 1 0.9636119 0.9636119 0.9636119
## 2 0.9636119 0.9636119
## 3 0.9636119
##
##
## Real Parameter p
## Group:sexFemale
## 2 3 4
```

```
## 1 0.4414296 0.9063971 0.705837
## 2          0.9063971 0.705837
## 3          0.705837
##
## Group:sexMale
##          2          3          4
## 1 0.4636091 0.9137224 0.72408
## 2          0.9137224 0.72408
## 3          0.72408
```

There are several things to notice in the above summary. First and most importantly, the number of parameters, AICc and -2LnL values are the same that we got above with MARK. Phew! I am kidding. Hopefully you are not surprised. I won't go into all of the details but as long as the data, DM and links are all the same, the results will be the same because it is the same piece of code (mark.exe) that is fitting the models. Differences between MARK and RMark (albeit usually small) have occurred and this question was asked often enough that the answers were put in a sticky message at the top of the RMark forum on phidot.org (<http://www.phidot.org/forum/viewtopic.php?f=21&t=2121>).

If you look at the real parameter estimates, you'll notice the differences in capture probability for the sexes and times (columns). Notice that the differences in the real parameters between the sexes across time are close but not all the same. While the sex effect is additive, it is additive on the logit scale which does not translate to additive differences on the real scale. The sex difference in survival is also obvious from the real values but why are they all the same? Didn't we have a difference due to the animals initial weight? Yes and what is presented is the survival for animals of average weight. To compute real survival estimates for individual animals or for a range of specific weight values, we will use the function `covariate.predictions` later in section 12.

6.4 Modifying and adding to the design data

The default design data can be modified (i.e., binning values) with arguments in `make.design.data` and the package contains a function `add.design.data` that can add design data covariates by binning some of the existing default values like `time`, `cohort` and `age`. However, the easiest and most flexible approach is to use base R functions to manipulate the design data. The one exception is if you want to merge a dataframe of covariates. You should not use the base R `merge` function without taking precautions to make sure the rows of the design data are not re-ordered. Doing so breaks the link between the PIMS and design data. You can always re-order them with the field `model.index` or use the function `merge.design.covariates` which does the merge and re-order for you.

The following is a typical question that I answer on the phidot forum for RMark. "My occasions are monthly and I can fit a time model but what I really want to fit is a seasonal or annual model. How do I do that with RMark?" The answer is to create a new field in the design data that puts the parameters into appropriate bins (intervals). Let's say you had 24 occasions with time values 1 to 24 and the first month was January and your seasons were every 3 months starting in January. Then the following would create season and year factor variables that you could use in your model.

```
time=1:24
year=floor((time-1)/12)
season=floor((time-year*12-1)/3)
year=factor(year)
season=factor(season,labels=c("Jan-Mar","Apr-June","July-Sept","Oct-Dec"))
year
## [1] 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
## Levels: 0 1
season
```

```
## [1] Jan-Mar Jan-Mar Jan-Mar Apr-June Apr-June Apr-June July-Sept July-Sept
## [9] July-Sept Oct-Dec Oct-Dec Oct-Dec Jan-Mar Jan-Mar Jan-Mar Apr-June
## [17] Apr-June Apr-June July-Sept July-Sept July-Sept Oct-Dec Oct-Dec Oct-Dec
## Levels: Jan-Mar Apr-June July-Sept Oct-Dec
```

Once those fields are added to the design data they can be used in a formula for that parameter. Note that had you tried to use `ddl$time` which is a factor variable you would have gotten an error because you can't do arithmetic on a factor variable even though it is a numeric variable.

```
floor(ddl$Phi$time/3)

## Warning in Ops.factor(ddl$Phi$time, 3): '/' not meaningful for factors

## [1] NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Instead you can use the numeric equivalents (e.g., `Time`) or convert the factor variable to numeric if that makes sense.

```
floor((ddl$Phi$Time+1)/3)

## [1] 0 0 1 0 1 1 0 0 1 0 1 1

floor(as.numeric(as.character(ddl$Phi$time))/3)

## [1] 0 0 1 0 1 1 0 0 1 0 1 1
```

Or you can use other base R functionality like `%in%` to create groups as shown below to add a variable `flood` which is an indicator for the 2 years in which there was a flood which may have affected survival.

```
ddl$Phi$flood=ifelse(ddl$Phi$time%in%c(1,3),1,0)
ddl$Phi[,-(1:2)]

##      group cohort age time occ.cohort Cohort Age Time sex flood
## 1 Female      1  0   1      1      0  0   0 Female    1
## 2 Female      1  1   2      1      0  1   1 Female    0
## 3 Female      1  2   3      1      0  2   2 Female    1
## 4 Female      2  0   2      2      1  0   1 Female    0
## 5 Female      2  1   3      2      1  1   2 Female    1
## 6 Female      3  0   3      3      2  0   2 Female    1
## 7 Male        1  0   1      1      0  0   0 Male      1
## 8 Male        1  1   2      1      0  1   1 Male      0
## 9 Male        1  2   3      1      0  2   2 Male      1
## 10 Male       2  0   2      2      1  0   1 Male      0
## 11 Male       2  1   3      2      1  1   2 Male      1
## 12 Male       3  0   3      3      2  0   2 Male      1
```

You can use any R commands to manipulate the design data but sometimes it is easiest to merge a dataframe of covariates into the design data. If you have many occasions and several weather covariates at those occasions that may have affected capture probability or survival, you don't really want to add these with individual R functions for each occasion. The RMark function `merge_design.covariates` was written for this application. I'll illustrate it with an example using a dataframe with the `flood` variable (with the name `Flood` or I'll get an error because `flood` already exists) and a temperature variable that will be added to the survival design data.

```
env.data=data.frame(time=1:3,temp=c(49,36,42),Flood=c(1,0,1))
ddl$Phi=merge_design.covariates(ddl$Phi,env.data)
ddl$Phi
```

```
##      time par.index model.index  group cohort age occ.cohort Cohort Age Time  sex
## 1      1         1         1  Female      1    0         1      0    0    0 Female
## 2      2         2         2  Female      1    1         1      0    1    1 Female
## 3      3         3         3  Female      1    2         1      0    2    2 Female
## 4      2         4         4  Female      2    0         2      1    0    1 Female
## 5      3         5         5  Female      2    1         2      1    1    2 Female
## 6      3         6         6  Female      3    0         3      2    0    2 Female
## 7      1         7         7   Male      1    0         1      0    0    0   Male
## 8      2         8         8   Male      1    1         1      0    1    1   Male
## 9      3         9         9   Male      1    2         1      0    2    2   Male
## 10     2        10        10   Male      2    0         2      1    0    1   Male
## 11     3        11        11   Male      2    1         2      1    1    2   Male
## 12     3        12        12   Male      3    0         3      2    0    2   Male
##      flood temp Flood
## 1      1    49     1
## 2      0    36     0
## 3      1    42     1
## 4      0    36     0
## 5      1    42     1
## 6      1    42     1
## 7      1    49     1
## 8      0    36     0
## 9      1    42     1
## 10     0    36     0
## 11     1    42     1
## 12     1    42     1
```

The function merged the dataframes where they had a common value of **time** because the default setting for the argument **bytime** is **TRUE**. Merging can also vary by **group** only or by **group** and **time** by changing the arguments **bygroup** and **bytime**. If this is not sufficiently flexible you can use the R **merge** function but make sure to retain the row order of the design data.

6.5 Age models

So far I have only mentioned that **age** and **Age** are created as part of the design data in CJS models but haven't talked about how you might use it to create models with age-specific estimates 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 our example data and randomly assign the initial age of the animals (possibly birds) as hatch-year and adult. The `ageclass` variable is then used to assign groups and design data.

```
set.seed(9481)
df$ageclass=factor(ifelse(runif(600)<0.5,0,1),labels=c("Hatch-Year","Adult"))
dp=process.data(df,model="CJS",groups=c("sex","ageclass"),age.var=2,initial.age=c(0,1))
ddl=make.design.data(dp)
ddl$Phi[,-c(1,2,6)]
```

##	group	cohort	age	occ.cohort	Cohort	Age	Time	sex	ageclass
## 1	FemaleHatch-Year	1	0	1	0	0	0	Female	Hatch-Year
## 2	FemaleHatch-Year	1	1	1	0	1	1	Female	Hatch-Year
## 3	FemaleHatch-Year	1	2	1	0	2	2	Female	Hatch-Year
## 4	FemaleHatch-Year	2	0	2	1	0	1	Female	Hatch-Year
## 5	FemaleHatch-Year	2	1	2	1	1	2	Female	Hatch-Year
## 6	FemaleHatch-Year	3	0	3	2	0	2	Female	Hatch-Year
## 7	MaleHatch-Year	1	0	1	0	0	0	Male	Hatch-Year
## 8	MaleHatch-Year	1	1	1	0	1	1	Male	Hatch-Year
## 9	MaleHatch-Year	1	2	1	0	2	2	Male	Hatch-Year
## 10	MaleHatch-Year	2	0	2	1	0	1	Male	Hatch-Year
## 11	MaleHatch-Year	2	1	2	1	1	2	Male	Hatch-Year
## 12	MaleHatch-Year	3	0	3	2	0	2	Male	Hatch-Year
## 13	FemaleAdult	1	1	1	0	1	0	Female	Adult
## 14	FemaleAdult	1	2	1	0	2	1	Female	Adult
## 15	FemaleAdult	1	3	1	0	3	2	Female	Adult
## 16	FemaleAdult	2	1	2	1	1	1	Female	Adult
## 17	FemaleAdult	2	2	2	1	2	2	Female	Adult
## 18	FemaleAdult	3	1	3	2	1	2	Female	Adult
## 19	MaleAdult	1	1	1	0	1	0	Male	Adult
## 20	MaleAdult	1	2	1	0	2	1	Male	Adult
## 21	MaleAdult	1	3	1	0	3	2	Male	Adult
## 22	MaleAdult	2	1	2	1	1	1	Male	Adult
## 23	MaleAdult	2	2	2	1	2	2	Male	Adult
## 24	MaleAdult	3	1	3	2	1	2	Male	Adult

Notice that we now have 4 groups and consequently 48 parameters in total with 24 for Phi and p each. The `age.var` argument (i.e., `age.var=2` in this case) defines the variable in the groups vector used to assign initial ages. The `initial.age` values are assigned in order of the `ageclass` factor variable. In this case they are in order, but remember factor variables are ordered alphanumerically by default. Had I assigned the values "Hatch-Year" and "Adult" rather than 0 and 1, then by default they would have been ordered with Adult first and then Hatch-Year. Also notice that the `initial.age.class` variable doesn't change but the `age` and `Age` variables start at different values for Hatch-Year and Adult classes and they change through time.

Now because adults are really anything that are 1 or older, I don't really know their true initial age beyond being at least 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:

```
ddl=make.design.data(dp,parameters=list(Phi=list(age.bins=c(0,1,4))),right=FALSE)
levels(ddl$Phi$age)=c("0","1Plus")
ddl$Phi[,-c(1,2,6)]
```

##	group	cohort	age	occ.cohort	Cohort	Age	Time	sex	ageclass
## 1	FemaleHatch-Year	1	0	1	0	0	0	Female	Hatch-Year
## 2	FemaleHatch-Year	1	1Plus	1	0	1	1	Female	Hatch-Year

## 3	FemaleHatch-Year	1	1Plus	1	0	2	2	Female	Hatch-Year
## 4	FemaleHatch-Year	2	0	2	1	0	1	Female	Hatch-Year
## 5	FemaleHatch-Year	2	1Plus	2	1	1	2	Female	Hatch-Year
## 6	FemaleHatch-Year	3	0	3	2	0	2	Female	Hatch-Year
## 7	MaleHatch-Year	1	0	1	0	0	0	Male	Hatch-Year
## 8	MaleHatch-Year	1	1Plus	1	0	1	1	Male	Hatch-Year
## 9	MaleHatch-Year	1	1Plus	1	0	2	2	Male	Hatch-Year
## 10	MaleHatch-Year	2	0	2	1	0	1	Male	Hatch-Year
## 11	MaleHatch-Year	2	1Plus	2	1	1	2	Male	Hatch-Year
## 12	MaleHatch-Year	3	0	3	2	0	2	Male	Hatch-Year
## 13	FemaleAdult	1	1Plus	1	0	1	0	Female	Adult
## 14	FemaleAdult	1	1Plus	1	0	2	1	Female	Adult
## 15	FemaleAdult	1	1Plus	1	0	3	2	Female	Adult
## 16	FemaleAdult	2	1Plus	2	1	1	1	Female	Adult
## 17	FemaleAdult	2	1Plus	2	1	2	2	Female	Adult
## 18	FemaleAdult	3	1Plus	3	2	1	2	Female	Adult
## 19	MaleAdult	1	1Plus	1	0	1	0	Male	Adult
## 20	MaleAdult	1	1Plus	1	0	2	1	Male	Adult
## 21	MaleAdult	1	1Plus	1	0	3	2	Male	Adult
## 22	MaleAdult	2	1Plus	2	1	1	1	Male	Adult
## 23	MaleAdult	2	1Plus	2	1	2	2	Male	Adult
## 24	MaleAdult	3	1Plus	3	2	1	2	Male	Adult

None of the animals will exceed age 4, so they were grouped into Hatch-Year and 1Plus using 4 as the upper limit and then the levels were reassigned to 0 and 1Plus. When I first constructed this example I used `levels(ddlPhiage)=c("0","1+")` but that constructed the variable name “age1+” which caused the code to fail. I need to investigate but it is best to avoid using symbols particularly at the end of labels. Use of an age variable for p may not make sense because they are all adults by the time of recapture but could be used if one thought that a “new adult” was more or less likely to be caught.

As an example, we will fit the model $\text{Phi}(\sim \text{age})\text{p}(\text{time})$

```
Phi.age=list(formula=~age)
p.time=list(formula=~time)
model=mark(dp,ddl,model.parameters=list(Phi=Phi.age,p=p.time))

##
## Output summary for CJS model
## Name : Phi(~age)p(~time)
##
## Npar : 5
## -2lnL: 1398.947
## AICc : 1409.006
##
## Beta
##          estimate      se      lcl      ucl
## Phi:(Intercept) 2.7534065 0.4153097 1.9393994 3.5674136
## Phi:age1Plus    -0.1870149 0.4978748 -1.1628495 0.7888197
## p:(Intercept)   -0.1946519 0.1503686 -0.4893744 0.1000706
## p:time3         2.5138119 0.2618854 2.0005166 3.0271073
## p:time4         1.1249136 0.1931753 0.7462899 1.5035373
##
##
## Real Parameter Phi
## Group:sexFemale.ageclassHatch-Year
```

```

##           1           2           3
## 1 0.9401054 0.9286670 0.9286670
## 2           0.9401054 0.9286670
## 3           0.9401054
##
## Group:sexMale.ageclassHatch-Year
##           1           2           3
## 1 0.9401054 0.9286670 0.9286670
## 2           0.9401054 0.9286670
## 3           0.9401054
##
## Group:sexFemale.ageclassAdult
##           1           2           3
## 1 0.9286667 0.9286667 0.9286667
## 2           0.9286667 0.9286667
## 3           0.9286667
##
## Group:sexMale.ageclassAdult
##           1           2           3
## 1 0.9286667 0.9286667 0.9286667
## 2           0.9286667 0.9286667
## 3           0.9286667
##
##
## Real Parameter p
## Group:sexFemale.ageclassHatch-Year
##           2           3           4
## 1 0.4514901 0.9104515 0.7171284
## 2           0.9104515 0.7171284
## 3           0.7171284
##
## Group:sexMale.ageclassHatch-Year
##           2           3           4
## 1 0.4514901 0.9104515 0.7171284
## 2           0.9104515 0.7171284
## 3           0.7171284
##
## Group:sexFemale.ageclassAdult
##           2           3           4
## 1 0.4514901 0.9104515 0.7171284
## 2           0.9104515 0.7171284
## 3           0.7171284
##
## Group:sexMale.ageclassAdult
##           2           3           4
## 1 0.4514901 0.9104515 0.7171284
## 2           0.9104515 0.7171284
## 3           0.7171284

```

What patterns do you see in the real parameters in PIM format? Can you construct the survival estimate from the betas for an Adult?

6.6 More complicated formula and models

When you are constructing models for your data, 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 may 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 so far. The most difficult part is translating your ideas about the biology and data collection into design data and formula. My standing joke, is that if the MARK interface is a hammer, the RMark interface is a nail gun. You can get things done more quickly but it is also easy to do damage quickly.

To build different or more complicated models, it is useful to learn more about R formulas. For example, if you don't want to use a treatment contrast for a factor variable but would rather have separate intercepts, you can do that by using either 0+ or -1+ as shown below:

```
# to avoid taking up space I'm turning off the output but showing the beta estimates
Phi.sex1=list(formula=~-1+sex)
Phi.sex2=list(formula=~0+sex)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex),output=FALSE)
mod0$results$AICc

## [1] 1522.122

summary(mod0)$beta

##           estimate      se      lcl      ucl
## Phi:(Intercept) 2.363514 0.2503927 1.8727448 2.854284
## Phi:sexMale     1.408346 0.6054646 0.2216351 2.595056
## p:(Intercept)   0.932743 0.0897772 0.7567797 1.108706

mod1=mark(dp,ddl,model.parameters=list(Phi=Phi.sex1),output=FALSE)
mod1$results$AICc

## [1] 1522.122

summary(mod1)$beta

##           estimate      se      lcl      ucl
## Phi:sexFemale 2.363514 0.2503927 1.8727448 2.854284
## Phi:sexMale   3.771860 0.6004762 2.5949269 4.948793
## p:(Intercept) 0.932743 0.0897772 0.7567796 1.108706

mod2=mark(dp,ddl,model.parameters=list(Phi=Phi.sex2),output=FALSE)
mod2$results$AICc

## [1] 1522.122

summary(mod2)$beta

##           estimate      se      lcl      ucl
## Phi:sexFemale 2.3635146 0.2503927 1.8727449 2.854284
## Phi:sexMale   3.7718609 0.6004765 2.5949271 4.948795
## p:(Intercept) 0.9327429 0.0897772 0.7567796 1.108706
```

The value for Females is the intercept in the first model and sexFemale in the other 2 models and they are all the same value. For males, the beta value sexMale is the increase for Males in the first model. For the other models the intercepts are separate and the value for Males is the Intercept + sexMale values in the first model.

Now let's expand the formula to include 2 factor variables in an additive model:

```
Phi.sex.age=list(formula=~sex+age)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.age),output=FALSE)
summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	2.0853411	0.3953772	1.3104017	2.860281
## Phi:sexMale	1.4173317	0.5787177	0.2830450	2.551618
## Phi:age1Plus	0.3976985	0.4907701	-0.5642110	1.359608
## p:(Intercept)	0.9344757	0.0883116	0.7613849	1.107567

What do the beta values represent? What set of factor levels are represented by the intercept? To help answer those questions it is useful to look at the DM.

```
mod0$design.matrix
```

	Phi:(Intercept)	Phi:sexMale	Phi:age1Plus	p:(Intercept)
## Phi gFemaleHatch-Year c1 a0 t1	"1"	"0"	"0"	"0"
## Phi gFemaleHatch-Year c1 a1 t2	"1"	"0"	"1"	"0"
## Phi gMaleHatch-Year c1 a0 t1	"1"	"1"	"0"	"0"
## Phi gMaleHatch-Year c1 a1 t2	"1"	"1"	"1"	"0"
## p gFemaleHatch-Year c1 a1 t2	"0"	"0"	"0"	"1"

You may be wondering why there are only 4 rows in the DM when there are 48 parameters. That will be explained later in section 6.10. Which group has only a single 1 in its row? What beta values are used to construct Adult Male survival?

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

```
Phi.age.x.sex=list(formula=~age*sex)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.age.x.sex),output=FALSE)
```

##

Note: only 4 parameters counted of 5 specified parameters

##

AICc and parameter count have been adjusted upward

```
summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	2.3111348	0.4924868	1.3458607	3.276409
## Phi:age1Plus	0.0984389	0.5860199	-1.0501600	1.247038
## Phi:sexMale	0.6089502	0.6978739	-0.7588826	1.976783
## Phi:age1Plus:sexMale	15.3324050	1645.3909000	-3209.6338000	3240.298600
## p:(Intercept)	0.9157811	0.0800235	0.7589350	1.072627

```
mod0$design.matrix

##                               Phi:(Intercept) Phi:age1Plus Phi:sexMale
## Phi gFemaleHatch-Year c1 a0 t1 "1"          "0"          "0"
## Phi gFemaleHatch-Year c1 a1 t2 "1"          "1"          "0"
## Phi gMaleHatch-Year c1 a0 t1   "1"          "0"          "1"
## Phi gMaleHatch-Year c1 a1 t2   "1"          "1"          "1"
## p gFemaleHatch-Year c1 a1 t2   "0"          "0"          "0"
##                               Phi:age1Plus:sexMale p:(Intercept)
## Phi gFemaleHatch-Year c1 a0 t1 "0"          "0"
## Phi gFemaleHatch-Year c1 a1 t2 "0"          "0"
## Phi gMaleHatch-Year c1 a0 t1   "0"          "0"
## Phi gMaleHatch-Year c1 a1 t2   "1"          "0"
## p gFemaleHatch-Year c1 a1 t2   "0"          "1"
```

What beta values do you use to construct Male-Adult survival? If you switch the formula to `~sex*age`, the order of the beta parameters changes and the name of the interaction variable changes. This only matters if you are using one model to provide initial values for another model as described in section 13.

If you want to fit a model in which the estimates are all computed separately, you can specify the formula as `~-1+sex:age` as shown below:

```
Phi.age.x.sex=list(formula=~-1+age:sex)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.age.x.sex),output=FALSE)

##
## Note: only 4 parameters counted of 5 specified parameters
##
## AICc and parameter count have been adjusted upward
summary(mod0)$beta

##               estimate          se          lcl          ucl
## Phi:age0:sexFemale    2.3111693    0.4925022    1.3458650    3.276474
## Phi:age1Plus:sexFemale 2.4095675    0.2973188    1.8268226    2.992312
## Phi:age0:sexMale      2.9201113    0.5142307    1.9122192    3.928003
## Phi:age1Plus:sexMale  18.6629800  1867.6349000 -3641.9014000  3679.227400
## p:(Intercept)         0.9157787    0.0800236    0.7589325    1.072625

mod0$results$AICc

## [1] 1522.981

# Note that the DM is an identity matrix
mod0$design.matrix

##                               Phi:age0:sexFemale Phi:age1Plus:sexFemale
## Phi gFemaleHatch-Year c1 a0 t1 "1"          "0"
## Phi gFemaleHatch-Year c1 a1 t2 "0"          "1"
## Phi gMaleHatch-Year c1 a0 t1   "0"          "0"
## Phi gMaleHatch-Year c1 a1 t2   "0"          "0"
## p gFemaleHatch-Year c1 a1 t2   "0"          "0"
##                               Phi:age0:sexMale Phi:age1Plus:sexMale p:(Intercept)
## Phi gFemaleHatch-Year c1 a0 t1 "0"          "0"          "0"
## Phi gFemaleHatch-Year c1 a1 t2 "0"          "0"          "0"
## Phi gMaleHatch-Year c1 a0 t1   "1"          "0"          "0"
## Phi gMaleHatch-Year c1 a1 t2   "0"          "1"          "0"
## p gFemaleHatch-Year c1 a1 t2   "0"          "0"          "1"
```

One advantage of using this approach is that you can use the sin link if the DM is an identity matrix. This is accomplished by using the link specification in the parameter list:

```
Phi.age.x.sex=list(formula=~-1+age:sex,link="sin")
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.age.x.sex),output=FALSE)
summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:age0:sexFemale	0.9606973	0.1410859	0.6841689	1.237226
## Phi:age1Plus:sexFemale	0.9883240	0.0817748	0.8280454	1.148602
## Phi:age0:sexMale	1.1144349	0.1133054	0.8923564	1.336513
## Phi:age1Plus:sexMale	1.5707963	0.2845030	1.0131705	2.128422
## p:(Intercept)	0.9157816	0.0800235	0.7589356	1.072628

```
mod0$results$AICc
## [1] 1522.981
```

Notice that the AICc values are the same, but for Phi the beta values are completely different because now the survival estimates are computed with the inverse sin link rather than the inverse logit link. The logit link is still being used for p, so its beta remains the same.

If you forgot to include the -1 in the formula, it would create a DM with 5 columns and the model is over-parameterized because it only takes 4 parameters to describe 4 groups. I'll go ahead and demonstrate this because it is useful to see what happens when this occurs.

```
Phi.age.x.sex=list(formula=~age:sex)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.age.x.sex),output=FALSE)
```


Note: only 4 parameters counted of 6 specified parameters
 ##
AICc and parameter count have been adjusted upward

```
summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	5.2386657	0.0000000	5.2386657	5.238666
## Phi:age0:sexFemale	-2.9274723	0.0000000	-2.9274723	-2.927472
## Phi:age1Plus:sexFemale	-2.8291388	0.0000000	-2.8291388	-2.829139
## Phi:age0:sexMale	-2.3186007	0.0000000	-2.3186007	-2.318601
## Phi:age1Plus:sexMale	13.0140200	1779.4325000	-3474.6737000	3500.701700
## p:(Intercept)	0.9157815	0.0800234	0.7589357	1.072627

```
mod0$design.matrix
```

	Phi:(Intercept)	Phi:age0:sexFemale	Phi:age1Plus:sexFemale	Phi:age0:sexMale
## Phi gFemaleHatch-Year c1 a0 t1	"1"	"1"		
## Phi gFemaleHatch-Year c1 a1 t2	"1"	"0"		
## Phi gMaleHatch-Year c1 a0 t1	"1"	"0"		
## Phi gMaleHatch-Year c1 a1 t2	"1"	"0"		
## p gFemaleHatch-Year c1 a1 t2	"0"	"0"		
## Phi gFemaleHatch-Year c1 a0 t1	"0"	"0"		
## Phi gFemaleHatch-Year c1 a1 t2	"1"	"0"		
## Phi gMaleHatch-Year c1 a0 t1	"0"	"1"		
## Phi gMaleHatch-Year c1 a1 t2	"0"	"0"		

```
## p gFemaleHatch-Year c1 a1 t2 "0" "0"
## Phi:age1Plus:sexMale p:(Intercept)
## Phi gFemaleHatch-Year c1 a0 t1 "0" "0"
## Phi gFemaleHatch-Year c1 a1 t2 "0" "0"
## Phi gMaleHatch-Year c1 a0 t1 "0" "0"
## Phi gMaleHatch-Year c1 a1 t2 "1" "0"
## p gFemaleHatch-Year c1 a1 t2 "0" "1"
```

```
summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	5.2386657	0.0000000	5.2386657	5.238666
## Phi:age0:sexFemale	-2.9274723	0.0000000	-2.9274723	-2.927472
## Phi:age1Plus:sexFemale	-2.8291388	0.0000000	-2.8291388	-2.829139
## Phi:age0:sexMale	-2.3186007	0.0000000	-2.3186007	-2.318601
## Phi:age1Plus:sexMale	13.0140200	1779.4325000	-3474.6737000	3500.701700
## p:(Intercept)	0.9157815	0.0800234	0.7589357	1.072627

```
mod0$design.matrix
```

```
## Phi:(Intercept) Phi:age0:sexFemale
## Phi gFemaleHatch-Year c1 a0 t1 "1" "1"
## Phi gFemaleHatch-Year c1 a1 t2 "1" "0"
## Phi gMaleHatch-Year c1 a0 t1 "1" "0"
## Phi gMaleHatch-Year c1 a1 t2 "1" "0"
## p gFemaleHatch-Year c1 a1 t2 "0" "0"
## Phi:age1Plus:sexFemale Phi:age0:sexMale
## Phi gFemaleHatch-Year c1 a0 t1 "0" "0"
## Phi gFemaleHatch-Year c1 a1 t2 "1" "0"
## Phi gMaleHatch-Year c1 a0 t1 "0" "1"
## Phi gMaleHatch-Year c1 a1 t2 "0" "0"
## p gFemaleHatch-Year c1 a1 t2 "0" "0"
## Phi:age1Plus:sexMale p:(Intercept)
## Phi gFemaleHatch-Year c1 a0 t1 "0" "0"
## Phi gFemaleHatch-Year c1 a1 t2 "0" "0"
## Phi gMaleHatch-Year c1 a0 t1 "0" "0"
## Phi gMaleHatch-Year c1 a1 t2 "1" "0"
## p gFemaleHatch-Year c1 a1 t2 "0" "1"
```

A few things are apparent that something went wrong. First, RMark reports that MARK has adjusted the count of parameters to 5 from the 6 specified because it thinks one of them is not identifiable. That is not always fool-proof and we will discuss more on that subject in section 13. A certain sign of problems is that all of the standard errors (se) are large for Phi. Finally, looking at the DM you'll see that there isn't a row for Phi with a single 1, so there isn't a true intercept level.

Now let's make this a little more complicated and say that we think the formula for Phi should be `~sex+time*age` but in reality we only released Hatch-year birds. To do so, I return to our original data format with groups only defined by sex but I still create the same age structure.

```
dp=process.data(df,model="CJS",groups="sex")
ddl=make.design.data(dp,parameters=list(Phi=list(age.bins=c(0,1,4))),right=FALSE)
levels(ddl$Phi$age)=c("0","1Plus")
```

The only thing that is different is that all of the animals are age 0 when they are released which means that for the survival interval from occasion 1 to occasion 2, I only have Hatch-year birds. For all subsequent intervals, I'll have birds in both age groups. Thus, I can estimate a different age-specific survival for all intervals except the first interval. If I specify the model as `~sex+time*age`, it will be over-parameterized because it will contain an estimate for Adults for the first interval as shown below:

```
Phi.sex.time.x.age=list(formula=~sex+time*age)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.time.x.age),output=FALSE)

##
## Note: only 5 parameters counted of 8 specified parameters
##
## AICc and parameter count have been adjusted upward

summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	1.7552953	0.2872773	1.1922317	2.318359
## Phi:sexMale	1.0766769	0.3978945	0.2968037	1.856550
## Phi:time2	16.8329720	1331.2766000	-2592.4692000	2626.135200
## Phi:time3	0.5293509	0.7828594	-1.0050537	2.063755
## Phi:age1Plus	-0.4772020	202.2375900	-396.8628900	395.908490
## Phi:time2:age1Plus	0.1282500	215.8114300	-422.8621500	423.118650
## Phi:time3:age1Plus	-0.7013433	202.2383400	-397.0884900	395.685800
## p:(Intercept)	1.0512124	0.0969865	0.8611189	1.241306

This type of over-parameterization is a little tougher to diagnose but notice that all of the parameters involving age have standard errors that are either 0 or large. What does the beta `Phi:age1Plus` represent in this model? It is the difference for `age1Plus` at time 1 which doesn't exist because we only had Hatch-Year birds at time 1; thus, it is not identifiable. The interactions `Phi:time2:age1Plus` and `Phi:time3:age1Plus` are measured relative to `Phi:age1Plus` so they are also affected with this structure.

We need to specify a model that excludes a parameter for `age1Plus` at time 1. One way to do that is to use `time:age` in the model instead of `time*age` because RMark will remove the columns in the DM that are empty (all 0s). An all 0 column only occurs when each time-age parameter is estimated separately. We also need to remove the intercept in this case but using -1 will not work because we also have a factor variable `sex` and as shown above -1+sex doesn't reduce the number of parameters but shifts to using 2 separate intercepts for the sexes. For this reason, I created the argument `remove.intercept` to forcibly remove the intercept as shown below:

```
Phi.sex.time.x.age=list(formula=~sex+time:age,remove.intercept=TRUE)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.time.x.age),output=FALSE)

##
## Note: only 5 parameters counted of 7 specified parameters
##
## AICc and parameter count have been adjusted upward

summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:sexMale	1.076691	0.3978830	0.2968406	1.856542
## Phi:time1:age0	1.755275	0.2872737	1.1922187	2.318332
## Phi:time2:age0	17.289727	1179.0329000	-2293.6148000	2328.194300
## Phi:time3:age0	2.284585	0.7572367	0.8004016	3.768769
## Phi:time2:age1Plus	19.102474	2189.2233000	-4271.7752000	4309.980100
## Phi:time3:age1Plus	1.106036	0.2717014	0.5735016	1.638571
## p:(Intercept)	1.051225	0.0969861	0.8611325	1.241318

Now you'll see that there are 3 parameters for age0 and 2 parameters (time 2 and 3) for age1Plus. So what is wrong with Phi:time2:age1Plus? It has a large positive value and the standard error is large and MARK has reported that one of the parameters has not been counted. Is it identifiable? In this case and the parameter is at a boundary because it is estimating Phi=1 for adults at time 2. Given that the other survival estimates are greater than 0.9 it is not surprising that one of the estimates hits a boundary at 1. The code MARK uses to locate non-identifiable parameters cannot discriminate between a parameter that is non-identifiable and one that is at a boundary especially with the logit link. Also, when parameters are at boundaries the usual method of estimating the standard error is not reliable. One way we can discern that is by dropping the sex effect and fitting the model with the logit and sin link.

```
Phi.sex.time.x.age=list(formula=~time:age,remove.intercept=TRUE,link="logit")
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.time.x.age),output=FALSE)

##
## Note: only 4 parameters counted of 6 specified parameters
##
## AICc and parameter count have been adjusted upward

summary(mod0)$beta

##
## estimate      se      lcl      ucl
## Phi:time1:age0  2.190530 2.695816e-01  1.6621498  2.718910
## Phi:time2:age0  33.314722 1.715999e-09 33.3147220 33.314722
## Phi:time3:age0   3.305939 1.484260e+00  0.3967890  6.215090
## Phi:time2:age1Plus 30.208662 0.000000e+00 30.2086620 30.208662
## Phi:time3:age1Plus 1.583838 2.831350e-01  1.0288931  2.138783
## p:(Intercept)   1.025445 1.011924e-01  0.8271082  1.223782

mod0$results$npar

## [1] 6

mod0$results$AICc

## [1] 1512.815

Phi.sex.time.x.age=list(formula=~time:age,remove.intercept=TRUE,link="sin")
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.time.x.age),output=FALSE)
summary(mod0)$beta

##
## estimate      se      lcl      ucl
## Phi:time1:age0  0.9252839 0.0810912 0.7663452 1.0842226
## Phi:time2:age0  1.5707963 0.1924055 1.1936816 1.9479111
## Phi:time3:age0  1.1924158 0.2741552 0.6550715 1.7297600
## Phi:time2:age1Plus 1.5707963 0.2387724 1.1028023 2.0387903
## Phi:time3:age1Plus 0.7201463 0.1064176 0.5115678 0.9287248
## p:(Intercept)   1.0254454 0.1011924 0.8271083 1.2237825

mod0$results$npar

## [1] 6

mod0$results$AICc

## [1] 1512.815
```

With the sin link, all of the parameters appear to be identifiable. Notice that the parameter count and AICc

are the same for both models. This is explained further in section 13.

Another approach is to use a dummy variable which is a numeric variable with values 0 and 1. I'll define a numeric variable called `male` and define it to be 0 when `sex="Female"` and 1 when `sex="Male"`. Then I'll fit the model `~-1+male+time:age`.

```
ddl$Phi$male=ifelse(ddl$Phi$sex=="Female",0,1)
Phi.sex.time.x.age=list(formula=~-1 + male+time:age)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.sex.time.x.age),output=FALSE)

##
## Note: only 5 parameters counted of 7 specified parameters
##
## AICc and parameter count have been adjusted upward

summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:male	1.076692	0.3978891	0.2968296	1.856555
## Phi:time1:age0	1.755266	0.2872730	1.1922109	2.318321
## Phi:time2:age0	18.145263	1681.5157000	-3277.6256000	3313.916200
## Phi:time3:age0	2.284597	0.7572445	0.8003971	3.768796
## Phi:time2:age1Plus	20.077865	2948.6933000	-5759.3610000	5799.516800
## Phi:time3:age1Plus	1.106070	0.2717092	0.5735197	1.638620
## p:(Intercept)	1.051221	0.0969863	0.8611279	1.241314

That only solved the extra parameter problem without using `remove.intercept` but did not solve the parameter boundary issue which is inherent to the data and the link function. If you happen to use `-1` and `remove.intercept`, it will give you an error message.

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 only for males because they were more likely to disperse. We can use the formula `~male:age` to fit that model. The intercept will be a constant female survival and there will be 2 parameters for male survival. Because the `male` variable has a value of 0 and 1, the age effect will only appear for males.

```
Phi.male.x.age=list(formula=~male:age)
mod0=mark(dp,ddl,model.parameters=list(Phi=Phi.male.x.age),output=FALSE)
summary(mod0)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	2.3472446	0.2480837	1.8610006	2.833489
## Phi:male:age0	2.9987505	3.9396659	-4.7229948	10.720496
## Phi:male:age1Plus	0.3876925	0.6280721	-0.8433289	1.618714
## p:(Intercept)	0.9466742	0.0912665	0.7677918	1.125557

6.7 Covariates: Design versus Individual

The difference between design and individual covariates is a point of confusion with many that first encounter RMark. A design covariate is a variable in the design data and it can be numeric or a factor variable. An individual covariate can only be a numeric variable and it is in the data with the capture history. A design covariate is the same for all animals (e.g., time) or the same for a group of animals (e.g., sex) whereas an individual covariate can have a different value for each animal. In the formula specification there is no difference in using a design covariate and an individual covariate (e.g., `weight`). However, there are differences in the computation of real parameters.

If you have recorded a factor variable like sex, then you should use it to define `groups` in `process.data`. Group variables expand the set of indices in the PIMs so there are separate indices and real parameters for males and for females. For design covariates, a real parameter is automatically calculated for each value of the design covariate because each value is associated with an index and each index represents a real parameter. That is not the case with individual covariates as we showed in the example that used `weight` in the model.

To demonstrate that difference instead of using sex as a factor variable, I'll create a numeric individual covariate named `male` which has values 0 for females and 1 for males. Then I'll build the model `~sex` for Phi:

```
# notice that I'm now using df rather than dp which was processed
# with groups
df$male=ifelse(df$sex=="Male",1,0)
model=mark(df,model.parameters=list(Phi=list(formula=~male)))

##
## Output summary for CJS model
## Name : Phi(~male)p(~1)
##
## Npar : 3
## -2lnL: 1516.098
## AICc : 1522.122
##
## Beta
##
##          estimate      se      lcl      ucl
## Phi:(Intercept) 2.363514 0.2503927 1.8727448 2.854284
## Phi:male         1.408346 0.6054651 0.2216341 2.595057
## p:(Intercept)   0.932743 0.0897772 0.7567797 1.108706
##
##
## Real Parameter Phi
##
##      1      2      3
## 1 0.95554 0.95554 0.95554
## 2      0.95554 0.95554
## 3      0.95554
##
##
## Real Parameter p
##
##      2      3      4
## 1 0.7176314 0.7176314 0.7176314
## 2      0.7176314 0.7176314
## 3      0.7176314
```

The real parameter estimates displayed are for the mean value of `male` which is 0.5 because there are 300 males and 300 females. If you wanted to get the survival estimates for females and males you would have to use `covariate.predictions` described in section 12.

If you have a numeric variable that has the same value for all animals (e.g., sampling effort over time) you can use it as a design covariate or an individual covariate. There are 2 advantages in using it as a design covariate:

- the model will be fitted more quickly because models with individual covariates require filling in the DM with the individual covariates to calculate the real parameters; whereas with design covariates there is only a single calculation of the DM, and

- a real parameter value will be automatically computed for each design covariate value.

However, you cannot compute real parameter estimates for values of the design covariate other than those in your design data. That is not the case with individual covariates which can be used to compute real parameter estimates for any value of the covariate. Thus, if you know that you want to compute real parameter estimates for a range of values other than what would be in the design data, then you should use an individual covariate. An example might be an environmental variable like temperature that might affect survival. Let's say that average temperature was calculated for each interval between occasions and you wanted to calculate predicting survival for a range of temperatures. You could use temperature as a design covariate and then do the calculations for survival with your own R code using the estimates of beta from the model and the temperature values. That is not particularly hard to do and you could use functions in RMark to help with that. Or you could define time-varying individual covariates `temp1`, `temp2`, `temp3` for each of the three intervals in our example data. There would only be 3 unique values of temperature and each value of `temp1` would be the same for each animal and likewise for `temp2` and `temp3`. As described in section 6.8 you would then use the value `~temp` in the formula for survival and you could predict survival for any value of temperature as described in section 12.

6.8 Time varying individual covariates

Time-varying individual covariates are a set of variables (one for each time) that are measured for each animal and that have a known value at each time. Time-varying individual covariates are not commonly used because you typically don't know the value at each time without capturing the animal on each occasion. Using design covariates like temperature described above is one exception. Another 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 $2 \dots k$ because it is the capture history value at occasion $1 \dots k-1$. 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 (time) 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 covariates in your data. Check the values of time in design data for the parameter being modeled with that time-varying individual covariates.

For example, let's assume that with our example data the first event is a capture and we want to model trap dependence for recapture probability for occasions 2 to 4. Instead of using the default value of `begin.time`, I'll use the value 1990. Thus, the recapture times will be labeled 1991 to 1993 and I'll want to label variables `td1991`, `td1992`, `td1993`. The variable `td1991` will have the value 1 if the animal was caught in 1990 and 0 otherwise. Likewise, `td1992` will have the value 1 if the animal was caught on occasion 1991 and 0 otherwise and `td1993` will have the value 1 if the animal was caught on occasion 1992 and 0 otherwise. When I create a formula, I'll use `~td` and RMark will recognize that `td` is not an individual covariate and is not in the design data. It then looks for individual covariates with the names `td1991`, `td1992`, `td1993` if it is used for `p` and for variables named `td1990`, `td1991`, `td1992` if used in formula for survival because the values of `time` in the design data differ. When it constructs the DM, it knows to put `td1991` in the rows for `p` that correspond to 1991 etc. An example of this is given in C.16 in Appendix C of the Cooch and White 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.

As an example here, I'll create some dummy temperature data for variables `temp1`, `temp2`, and `temp3` for the intervals for survival.

```

df$temp1=rep(10.1,600)
df$temp2=rep(13.5,600)
df$temp3=rep(8.5,600)
temp.model=mark(df,model.parameters=list(Phi=list(formula=~temp)))

##
## Output summary for CJS model
## Name : Phi(~temp)p(~1)
##
## Npar : 3
## -2lnL: 1510.786
## AICc : 1516.81
##
## Beta
##
## estimate se lcl ucl
## Phi:(Intercept) -2.3675888 1.3849757 -5.0821412 0.3469635
## Phi:temp 0.4853479 0.1451488 0.2008563 0.7698395
## p:(Intercept) 1.0799428 0.1012819 0.8814303 1.2784553
##
##
## Real Parameter Phi
##
## 1 2 3
## 1 0.9265202 0.9850002 0.8529425
## 2 0.9850002 0.8529425
## 3 0.8529425
##
##
## Real Parameter p
##
## 2 3 4
## 1 0.7464832 0.7464832 0.7464832
## 2 0.7464832 0.7464832
## 3 0.7464832

```

The survival estimates for each value of temperature are shown in the real parameter estimates for time1, time2, time3. I made up these temperature values, so was it luck or is there another explanation why temperature appears to be significant? You could compute the real values over a range of temperatures using:

```

phi=coef(temp.model)[1:2,]
plogis(phi$estimate[1]+c(10.1,13.5,8.5)*phi$estimate[2])

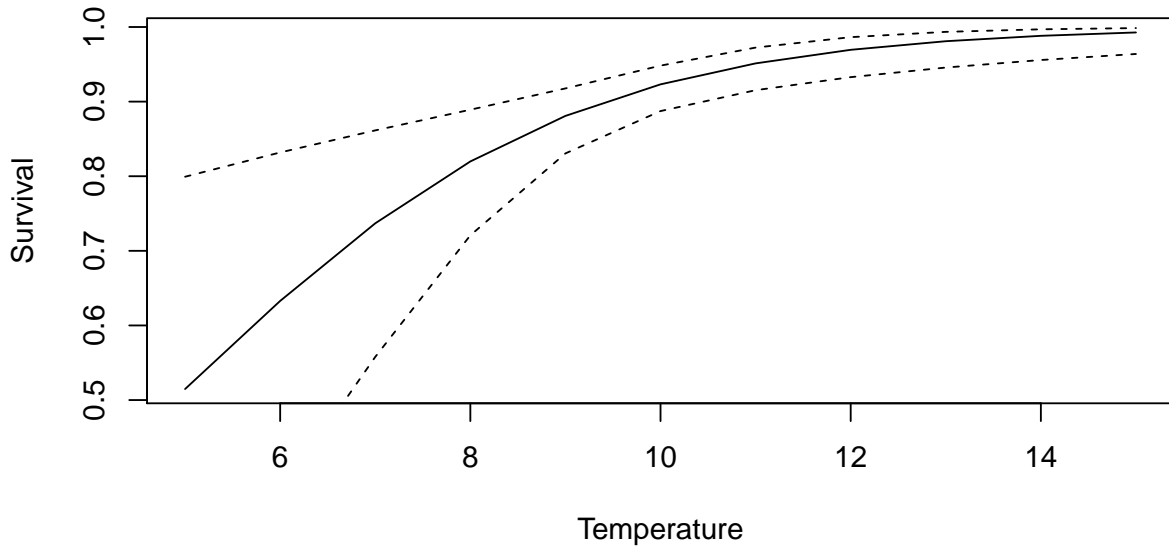
## [1] 0.9265202 0.9850002 0.8529425

```

```

logit.values=phi$estimate[1]+(5:15)*phi$estimate[2]
deriv=matrix(1,ncol=2,nrow=11)
deriv[,2]=5:15
std.errors=sqrt(diag(deriv%*%temp.model$results$beta.vcv[1:2,1:2]*%*t(deriv)))
lcl.logit=logit.values-1.96*std.errors
ucl.logit=logit.values+1.96*std.errors
plot(5:15,plogis(logit.values),type="l",xlab="Temperature",ylab="Survival")
lines(5:15,plogis(lcl.logit),lty=2)
lines(5:15,plogis(ucl.logit),lty=2)

```



Something similar to the above calculations could have also been used had I treated temperature as a design covariate. In section 12, I'll show how the same calculations could be made with `covariate.predictions`.

6.9 Parameter sharing

Most types of parameters in capture-recapture models represent different quantities and thus you would never have the same beta parameter value for different types of parameters (e.g., the a beta column for Phi would not be used for p). However, there are exceptions and RMark has made allowance for parameters sharing the same column of the DM. The sharing capability is not completely general and there may be circumstances in which you cannot construct some models with RMark. The only place I have seen this so far is with some of the occupancy models.

When pairs of different types of parameters measure a similar quantity, one of the parameters is treated as the dominant parameter and the other parameter is subsumed under the dominant parameter. For example, in the Closed capture models, `p` is initial capture probability and `c` is recapture probability. It is quite reasonable to fit a model with `p=c`. For Closed models, I have chosen to make `p` the dominant parameter in the `p&c` pairing. The parameter `c` is shared with `p` by setting the argument `share=TRUE` in the parameter specification for `p` and the parameter specification for `c` is not provided. For example, `p.c=list(formula=~1,share=TRUE)` will create a model in which there is constant capture probability which equals the recapture probability as well. Sometimes it can be useful to have a model in which recapture probability differs from capture probability in an additive model. For example, `p.time.c=list(formula=~Time+c,share=TRUE)` describes a model in which capture probability varies with a linear trend over time and recapture probability also varies with the same slope but with an additive difference. This works because when `share=TRUE` RMark combines the design data for `p` with the design data for `c` and adds a column `c` to the combined design data which has value 0 for the indices (rows) representing `p` and a value of 1 for the indices representing `c`. In constructing this model I realized that Time is created separately for each parameter such that the origin is the first time for the parameter. That creates incorrect values of Time when the values are shared with `p` and `c`. The same thing will happen with any numeric variable Time or Age when the parameters start at different times and are shared, so be aware. It can be easily fixed by setting `ddlcTime=ddlcTime+1`. The combined design data is temporary and is not saved outside of the model. Below is an example using the `edwards.eberhardt` rabbit data that accompanies both MARK and RMark.

```

data(edwards.eberhardt)
dp.ee=process.data(edwards.eberhardt,model="Closed")
ddl.ee=make.design.data(dp.ee)
# first fit the default model with constant but
# different p and c
model=mark(dp.ee,ddl.ee,output=FALSE)
summary(model)$beta

##              estimate      se      lcl      ucl
## p:(Intercept) -2.352694 0.2716165 -2.885063 -1.820326
## c:(Intercept) -2.429084 0.1284003 -2.680749 -2.177420
## f0:(Intercept)  2.884345 0.6045293  1.699468  4.069223

# next fit constant p=c
p.c=list(formula=~1,share=TRUE)
model=mark(dp.ee,ddl.ee,model.parameters=list(p=p.c),output=FALSE)
summary(model)$beta

##              estimate      se      lcl      ucl
## p:(Intercept) -2.416076 0.1171742 -2.645738 -2.186415
## f0:(Intercept)  3.008588 0.3395371  2.343095  3.674081

# next fit constant p(Time) c(Time) where logit(c(Time))=logit(p(Time))+beta_c
ddl.ee$c$Time=ddl.ee$c$Time+1
p.Time.c=list(formula=~Time+c,share=TRUE)
model=mark(dp.ee,ddl.ee,model.parameters=list(p=p.Time.c))

##
## Output summary for Closed model
## Name : p(~Time + c)c()f0(~1)
##
## Npar : 4
## -2lnL: 358.8021
## AICc : 366.8315
##
## Beta
##              estimate      se      lcl      ucl
## p:(Intercept) -19.9102620 0.4479701 -20.7882830 -19.0322400
## p:Time         -0.1031804 0.0163218  -0.1351711  -0.0711897
## p:c            18.5409030 0.4568708  17.6454360  19.4363690
## f0:(Intercept) 22.0910970 0.4198267  21.2682370  22.9139580
##
##
## Real Parameter p
##              1              2              3              4              5              6
## 2.254672e-09 2.033633e-09 1.834264e-09 1.654441e-09 1.492247e-09 1.345953e-09
##              7              8              9             10             11             12
## 1.214002e-09 1.094986e-09 9.876382e-10 8.908143e-10 8.034827e-10 7.247127e-10
##             13             14             15             16             17             18
## 6.536649e-10 5.895824e-10 5.317822e-10 4.796486e-10 4.326259e-10 3.902131e-10
##
##
## Real Parameter c
##              2              3              4              5              6              7              8              9
## 0.186557 0.1714025 0.1572412 0.1440464 0.1317858 0.1204219 0.1099138 0.1002182

```

```
##          10          11          12          13          14          15          16          17
## 0.0912902 0.0830841 0.0755543 0.0686558 0.0623447 0.0565785 0.0513164 0.0465196
##          18
## 0.0421513
##
##
## Real Parameter f0
##          1
## 3926827000
```

Clearly that is not a useful model and a better model has separate intercept and slopes for Time:

```
p.Time=list(formula=~Time)
c.Time=list(formula=~Time)
model=mark(dp.ee,ddl.ee,model.parameters=list(p=p.Time,c=c.Time))

##
## Output summary for Closed model
## Name : p(~Time)c(~Time)f0(~1)
##
## Npar : 5
## -2lnL: 354.0074
## AICc : 364.0515
##
## Beta
##          estimate          se          lcl          ucl
## p:(Intercept) -2.3712300 0.2143091 -2.7912758 -1.9511842
## p:Time         0.0813806 0.0464166 -0.0095960 0.1723572
## c:(Intercept) -1.1114614 0.2991856 -1.6978652 -0.5250575
## c:Time        -0.1307622 0.0297183 -0.1890099 -0.0725144
## f0:(Intercept) 0.8847648 1.5782832 -2.2086703 3.9781999
##
##
## Real Parameter p
##          1          2          3          4          5          6          7          8
## 0.085393 0.0919671 0.0989926 0.1064917 0.1144868 0.1229995 0.1320508 0.1416606
##          9          10         11         12         13         14         15         16
## 0.1518473 0.1626278 0.1740166 0.1860258 0.1986645 0.2119382 0.2258489 0.240394
##          17         18
## 0.2555667 0.2713549
##
##
## Real Parameter c
##          2          3          4          5          6          7          8          9
## 0.2240492 0.2021379 0.1818672 0.1632135 0.1461313 0.130558 0.116418 0.103627
##          10         11         12         13         14         15         16         17
## 0.0920949 0.081729 0.0724369 0.0641274 0.0567129 0.0501097 0.0442393 0.0390284
##          18
## 0.0344091
##
##
## Real Parameter f0
##          1
## 2.422415
```

The models with parameters that can be shared and the parameter pairings (dominant parameter first) are listed below. In each case a column with the name of the non-dominant parameter is added to the combined design data so additive differences and other types of formulas can be specified:

6.10 PIM simplification

For some capture-recapture studies with many occasions and many groups, the size of the DM becomes very large because RMark by default uses the all-different PIM formulation which has a different index for each parameter and each index requires a row in the DM. This is necessary so as to not restrict the set of models that can be built but it has a downside. As the size of the DM increases the amount of computations and length of time increases for mark.exe to fit models to the data. However, a more severe consequence is that mark.exe computes the variance-covariance matrix for the real parameters which has n^2 elements where n is the number of rows in the DM. If n gets very large, there may be insufficient memory to compute the real parameter 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. Our example has only 48 rows in the DM, but it is easy to create examples with 10,000 rows in the DM 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.

With our example data using the four sex-age groups, if the model was $\text{Phi}(\sim 1)\text{p}(\sim 1)$ then Phi and p are constant so each of the 24 real Phis and 24 real p's have the same value. Thus we really only need 2 real parameters. In the MARK interface that could be done by changing the PIMs as we showed above. With RMark that simplification of the PIMS is done automatically after the DM is built and before the input file is constructed for MARK. The DM is constructed and then the PIMs are simplified by finding the unique rows in the DM. As part of that uniqueness, any fixed values for the real parameters are also included. The PIM coding is simplified to 2 indices, For our example, the original indices 1-24 for Phi are assigned to 1 and indices 25-48 for p are assigned to 2. We can see this in the model results as shown below where the DM used with MARK is a 2x2 identity matrix:

```
model=mark(dp,ddl,output=FALSE)
model$design.matrix

##                               Phi:(Intercept) p:(Intercept)
## Phi gFemale c1 a0 t1 "1"                               "0"
## p gFemale c1 a1 t2  "0"                               "1"

# show the PIM translation vector
model$simplify$pim.translation

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

You can also use the function PIMS to see the simplified and non-simplified pims for a particular type of parameter:

```
PIMS(model,"p")

## group = sexFemale
##      2  3  4
## 1    2  2  2
## 2      2  2
## 3        2
## group = sexMale
##      2  3  4
## 1    2  2  2
## 2      2  2
```

```
## 3      2

PIMS(model,"p",simplified=FALSE)

## group = sexFemale
##      2  3  4
## 1 13 14 15
## 2      16 17
## 3      18
## group = sexMale
##      2  3  4
## 1 19 20 21
## 2      22 23
## 3      24
```

RMark maintains the original indices with the simplified indices for model averaging real parameters. As the models change, the number and indices of the simplified parameters change to match the model complexity, but the underlying original typically all-different indices remain the same. Because the simplified indices differ across models they would not be useful for averaging across models. The original non-simplified indices are also in `model.index` in the design data. The one side-effect of simplification is that the real parameter labels in the output file produced by MARK are not useful. However, labels in the RMark summaries can be useful although the pim formatted output provided by summary is typically more useful. The real labels are pasted values of group (g), cohort (c), time (t), and age (a).

7 Fixing Real Parameter Values

Occasionally it is necessary or appropriate to fix the values of one or more real parameters. You cannot fix the values of beta parameters. There are several ways to fix real parameter values in RMark and this has evolved with its development. My original approach was to specify an argument named `fixed` in the parameter specification list and this is still allowed but I'll first describe the easiest approach that was added in the last year. It is described in the sticky note at the top of the RMark sub-forum (<http://www.phidot.org/forum/viewtopic.php?f=21&t=2887>).

In many cases the parameters are fixed due to the structure of the data or sampling. For example, animals may only be released and marked every other year but are recaptured (resighted) every year or sampling may have not been conducted in an area some years. If you want to fix some parameter values across all models, then it is easiest to do that by adding a field called `fix` to the design data. You assign the value NA to `fix` for any real parameter that should be estimated and for those that should be fixed, assign the real value to `fix`. Doing so will fix the real parameters to those values for all models that use those design data. Below is an example that uses the example data and fixes p for time 3 to 0.

```
ddl$Phi$fix=ifelse(ddl$Phi$time==2,1,NA)
model=mark(dp,ddl)

##
## Output summary for CJS model
## Name : Phi(~1)p(~1)
##
## Npar : 2
## -2lnL: 1506.678
## AICc : 1510.69
##
## Beta
##      estimate      se      lcl      ucl
## Phi:(Intercept) 2.091182 0.2163291 1.6671768 2.515187
```



```
## p:(Intercept)    1.008537 0.0896932 0.8327383 1.184336
##
##
## Real Parameter Phi
## Group:sexFemale
##           1 2           3
## 1 0.8900431 0.8900431
## 2           0.8900431
## 3           0.8900431
##
## Group:sexMale
##           1 2           3
## 1 0.8900431 0.8900431
## 2           0.8900431
## 3           0.8900431
##
##
## Real Parameter p
## Group:sexFemale
##           2           3           4
## 1 0.7327337 0.7327337 0.7327337
## 2           0.7327337 0.7327337
## 3           0.7327337
##
## Group:sexMale
##           2           3           4
## 1 0.7327337 0.7327337 0.7327337
## 2           0.7327337 0.7327337
## 3           0.7327337
```

Fixed parameters do not show up in the summary by default. If you add the argument `show.fixed=TRUE` they will appear with their fixed values.

```
summary(model,show.fixed=TRUE)

## Output summary for CJS model
## Name : Phi(~1)p(~1)
##
## Npar : 2
## -2lnL: 1506.678
## AICc : 1510.69
##
## Beta
##           estimate          se          lcl          ucl
## Phi:(Intercept) 2.091182 0.2163291 1.6671768 2.515187
## p:(Intercept)   1.008537 0.0896932 0.8327383 1.184336
##
##
## Real Parameter Phi
## Group:sexFemale
##           1 2           3
## 1 0.8900431 1 0.8900431
## 2           1 0.8900431
## 3           0.8900431
```

```
##
## Group:sexMale
##      1 2      3
## 1 0.8900431 1 0.8900431
## 2      1 0.8900431
## 3      0.8900431
##
##
## Real Parameter p
## Group:sexFemale
##      2      3      4
## 1 0.7327337 0.7327337 0.7327337
## 2      0.7327337 0.7327337
## 3      0.7327337
##
## Group:sexMale
##      2      3      4
## 1 0.7327337 0.7327337 0.7327337
## 2      0.7327337 0.7327337
## 3      0.7327337
```

Another way to fix real parameters to their default value is to delete design data records. While this is possible to do, it is better to use the `fix` field because deleting design data records does not work with `mlogit` parameters and can give incorrect results.

Real parameters can be fixed for a particular model using the the original method of specifying fixed real parameters by adding the argument `fixed` to the list specification of the parameter. For example, `p.1=list(formula=~1, fixed=1)` would set all of the values of `p` to 1 turning a CJS model into a known fate model. An example where this is useful is the live-dead models, where one may set the fidelity parameter `F=1` when capture and dead recovery areas are the same. There are also list formulations for the `fixed` argument including `fixed=list(time=..., value=...)`, `fixed=list(cohort=..., value=...)`, `fixed=list(age=..., value=...)` where specific values of `time`, `cohort` and `age` are specified and `value` is the fixed real parameter value. The most general approach is to use `fixed=list(index=...,value=...)` where `index` is a subset of the `par.index` field in the design data for a parameter and `value` is either a single fixed parameter value or a vector of fixed parameter values that matches the length of the `index` vector. Below are two examples in which `p` is set to 1 for time 3.

```
#using time
ddl$Phi$fix=NULL
p.time=list(formula=~time,fixed=list(time=3,value=1))
model=mark(dp,ddl,model.parameters=list(p=p.time),output=FALSE)
summary(model,show.fixed=TRUE)

## Output summary for CJS model
## Name : Phi(~1)p(~time)
##
## Npar : 3
## -2lnL: 32386.86
## AICc : 32392.88
##
## Beta
##      estimate      se      lcl      ucl
## Phi:(Intercept) 2.3092240 0.1483552 2.0184478 2.6000001
## p:(Intercept) -0.1983421 0.1537991 -0.4997885 0.1031042
## p:time4      1.2076885 0.1984092 0.8188065 1.5965705
```

```
##
##
## Real Parameter Phi
## Group:sexFemale
##           1           2           3
## 1 0.9096381 0.9096381 0.9096381
## 2           0.9096381 0.9096381
## 3           0.9096381
##
## Group:sexMale
##           1           2           3
## 1 0.9096381 0.9096381 0.9096381
## 2           0.9096381 0.9096381
## 3           0.9096381
##
##
## Real Parameter p
## Group:sexFemale
##           2 3           4
## 1 0.4505764 1 0.7328922
## 2           1 0.7328922
## 3           0.7328922
##
## Group:sexMale
##           2 3           4
## 1 0.4505764 1 0.7328922
## 2           1 0.7328922
## 3           0.7328922

# using index; many of the examples are shown using something like
# time3=as.numeric(rownames(ddl$p[ddl$p$time==3,]))
# because those examples were written before I added par.index field
# to code
time3=ddl$p$par.index[ddl$p$time==3]
p.time=list(formula=~time,fixed=list(index=time3,value=1))
model=mark(dp,ddl,model.parameters=list(p=p.time),output=FALSE)
summary(model,show.fixed=TRUE)

## Output summary for CJS model
## Name : Phi(~1)p(~time)
##
## Npar : 3
## -2lnL: 32386.86
## AICc : 32392.88
##
## Beta
##           estimate      se      lcl      ucl
## Phi:(Intercept) 2.3092255 0.1483553 2.0184492 2.6000018
## p:(Intercept)   -0.1983408 0.1537990 -0.4997868 0.1031052
## p:time4          1.2076862 0.1984088 0.8188049 1.5965675
##
##
## Real Parameter Phi
## Group:sexFemale
```

```
##           1           2           3
## 1 0.9096382 0.9096382 0.9096382
## 2           0.9096382 0.9096382
## 3           0.9096382
##
## Group:sexMale
##           1           2           3
## 1 0.9096382 0.9096382 0.9096382
## 2           0.9096382 0.9096382
## 3           0.9096382
##
##
## Real Parameter p
## Group:sexFemale
##           2 3           4
## 1 0.4505767 1 0.732892
## 2           1 0.732892
## 3           0.732892
##
## Group:sexMale
##           2 3           4
## 1 0.4505767 1 0.732892
## 2           1 0.732892
## 3           0.732892
```

While this is still a valid approach to fix the parameters for specific models, if you are going to fix the real parameter values for all models it is cleaner and easier to add the `fix` field to the design data as described above.

8 Scripting and Model Selection

So far I have only shown examples of running a single model. Next I'll describe how to set up a function to analyze a set of models for your analysis. But before I get there it is important to keep in mind the following points:

- Too many models can be a bad thing! Beware of paralysis of analysis!
- Too much data dredging can lead you to a nonsensical “best” model.
- A nonsensical “best” model is still nonsensical and not useful!

Because RMark makes it fairly easy to construct models, it is easy to get carried away. With models containing many parameters, it is fairly easy to specify several thousand models if you don't do some critical thinking.

You may have already discovered that it is quite easy to make mistakes while you are typing in the R console by forgetting a comma or parenthesis here and there. Also, it may be clear that if you created many models you would have to devise a bunch of different names for the model object and after awhile the formulas would clutter 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 a development environment like Rstudio. With an external environment like Rstudio you get:

- syntax checking and auto-completion; matching braces and parentheses

- built-in help for R
- easily running all or part of the script in the R console from the script editor window

With scripts you can add comments. Comments 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.

If you want to take the next step beyond scripts investigate reproducible research tools. Those tools allow you to interleave the text for your paper with your R code and the results from the R code into tables, figures and computed values embedded directly into the text. A reproducible research document:

- Reduces the chances of making mistakes transcribing results into a paper
- Makes it easy to reproduce the entire paper once you receive review comments or if you happened to find errors in your data
- Provides documentation for all of the calculations in your document when you get questions from reviewers or interested researchers
- Provides a learning tool for others who can follow how you did your analysis.

The easiest reproducible research tool to learn is `rmarkdown` with the `knitr` package which can output HTML and Word files and a PDF if you have a \LaTeX system. Both `rmarkdown` and `knitr` are integrated into the Rstudio development environment. If you do have \LaTeX then you should consider using `knitr` or Sweave to construct your reproducible documents. \LaTeX is far more capable than `rmarkdown` but it is harder to learn and you cannot create a Word document. I use the \LaTeX gui interface for \LaTeX which helps reduce the burden of learning \LaTeX . This document was written with \LaTeX for \LaTeX with `knitr`. The code file accompanying this document includes the various code chunks contained in the \LaTeX document.

I also need to introduce functions which can be used in scripts. You are already using base R functions and functions from R packages; however, you may never have written a function. To define a function from code in a script all you need to do is:

- wrap the script code in braces `{}`,
- add `somename= function()` on the first line before the left brace `{`, where `somename` will be the name of your function,
- add a line before the right `}` with a **return** function to return any object or list of objects that you want to return to the calling environment, and
- you can also optionally add arguments in the `()` after function which allows you to pass values to the function; however, for our functions we will not use arguments.

Once your function is defined you need to call your function to execute the code in the function. It is similar to copying and pasting your script into the R console. However, functions are different in a couple of ways which make them very useful. First, the objects in the function “workspace” that would be returned with the `ls()` function are only those that you define in your function. That will be useful below when we create sets of models. If you want to see this for yourself type `ls()` in the R console with a non-empty workspace and see the names of the objects in your workspace. Then define the following very short function by typing the following into your R console: `myf=function() { x=1; return(ls()) }`. If you were then to type `myf` into the console, R will show you the contents of the function `myf` but doesn’t call it. To call the function you need to type `myf()` and it will show you a vector containing the name “**x**” which is all that is contained in the function workspace. 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 the function used an object `y` that did not exist in `myf`

but existed in the parent environment (typically the workspace) of `myf`, it would find `y` and use it. In other words, if you create an object in a script (e.g., processed data list) outside of the function, 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 fully 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.

So why create a function? You create a function so that you can create a set of formulas for each parameter in the model and then use functions `create.model.list` and `mark.wrapper` to create the set of models from all combinations of the parameter specifications, run each and return a list with each of the models and a model selection table called `model.table`. Because the call to `create.model.list` is embedded in the function, it will only retrieve parameter specifications with names like `parameter.xxx` (e.g., `Phi.time`) defined within the function that are a list containing an object named `formula`. That was done to make sure it was not confused other objects with that naming structure (e.g., `Phi.0=0`). If for some reason you do not see all of the model combinations attempted, check to make sure each is a list with a formula. In my first attempt I spelled formula as `formula` and I didn't get all combinations. Below is an example which could be stored in a script.

```
# read in data
df=import.chdata("example.txt",field.types=c("f","n"))
# create fake age data and make it a factor variable
set.seed(9481)
df$ageclass=factor(ifelse(runif(600)<0.5,0,1),labels=c("Hatch-Year","Adult"))
# process data for CJS model with sex and age groups
dp=process.data(df,model="CJS",groups=c("sex","ageclass"),age.var=2,initial.age=c(0,1))
# create design data with age bins for Phi
ddl=make.design.data(dp,parameters=list(Phi=list(age.bins=c(0,1,4))),right=FALSE)
levels(ddl$Phi$age)=c("0","1Plus")
# create analysis function
do_analysis=function()
{
  # create formulas for Phi
  Phi.dot=list(formula=~1)
  Phi.sex=list(formula=~sex)
  Phi.sex.age=list(formula=~sex+age)
  Phi.sex.weight=list(formula=~sex+weight)
  #create formulas for p
  p.dot=list(formula=~1)
  p.time=list(formula=~time)
  # create all combinations (8 models)
  cml=create.model.list("CJS")
  # run all all 8 models and return as a list with class marklist
  results=mark.wrapper(cml,data=dp,ddl=ddl,output=FALSE,silent=TRUE)
  return(results)
}
# Now call the function to run the models and return the results
# stored into example.results
example.results=do_analysis()
# Show the model selection table
example.results
```

##	model	npar	AICc	DeltaAICc	weight	Deviance
## 8	Phi(~sex + weight)p(~time)	6	1393.198	0.00000	0.9648913852	1381.11450
## 6	Phi(~sex)p(~time)	5	1400.498	7.29986	0.0250803711	51.89973
## 4	Phi(~sex + age)p(~time)	6	1402.521	9.32290	0.0091208732	51.89883
## 2	Phi(~1)p(~time)	4	1407.137	13.93844	0.0009073705	60.55824

```
## 7      Phi(~sex + weight)p(~1)      4 1515.238 122.03974 0.0000000000 1507.19810
## 5          Phi(~sex)p(~1)          3 1522.122 128.92373 0.0000000000 177.55939
## 3      Phi(~sex + age)p(~1)      4 1523.536 130.33764 0.0000000000 176.95740
## 1          Phi(~1)p(~1)          2 1529.047 135.84891 0.0000000000 186.49653
```

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! Be aware that many of the examples in the RMark help files do not use this approach because they were written before I developed `mark.wrapper`.

Now let's look at the results. What is shown is a model selection table that is similar to what the MARK interface provides. For each fitted model, listed in order of ascending AICc value, the following is given:

- model number: element number in the marklist that contains that model
- model name using the formulas for each parameter
- npar: number of beta parameters
- AICc: small sample corrected Akaike's Information Criterion
- DeltaAICc: difference in AICc from best model
- weight: model weight based on DeltaAICc value
- Deviance: residual deviance (null deviance - explained deviance from model)

One obvious aspect of this table and a point of confusion and questions is the deviance value that is wildly different for models with and without individual covariates. The reason is that null deviance is 0 for models with individual covariates and it has a non-zero value for models without covariates. You will never see this occur with models created in the MARK interface because when you define the project to the MARK interface with an individual covariate, all models will be treated as if they have an individual covariate even if one is not used in the DM. With RMark each input file is created based on the information provided in the formulas. If there is no individual covariate in the formulas, then none are described in the inp file passed to mark.exe so it is treated as if there were no individual covariates. This has the effect shown above and the positive effect that models without covariates will run more quickly in mark.exe. It runs more quickly because mark.exe accumulates similar capture histories and only computes the likelihood with the unique capture histories and uses the accumulated freq value in computing the likelihood ($\text{freq} * \ln(\text{ch})$); whereas, with individual covariates it will not accumulate like capture histories if the inp file is defined with individual covariates. Note that RMark does this accumulation with or without covariates prior to creating the inp file and thus gets the gain in speed without relying on mark.exe. Now, if it bothers you to see the deviance value, you can use the following code to switch the model selection table to show $-2\ln L$ ($2 * \text{negative log-likelihood}$) which is used to compute AICc which are not wildly different:

```
example.results$model.table=model.table(example.results,use.ln1=TRUE)
example.results

##          model npar      AICc DeltaAICc      weight  Neg2LnL
## 8 Phi(~sex + weight)p(~time)      6 1393.198      0.00000 0.9648913852 1381.114
## 6          Phi(~sex)p(~time)      5 1400.498      7.29986 0.0250803711 1390.438
## 4      Phi(~sex + age)p(~time)      6 1402.521      9.32290 0.0091208732 1390.437
## 2          Phi(~1)p(~time)      4 1407.137     13.93844 0.0009073705 1399.097
```

## 7	Phi(~sex + weight)p(~1)	4	1515.238	122.03974	0.0000000000	1507.198
## 5	Phi(~sex)p(~1)	3	1522.122	128.92373	0.0000000000	1516.098
## 3	Phi(~sex + age)p(~1)	4	1523.536	130.33764	0.0000000000	1515.496
## 1	Phi(~1)p(~1)	2	1529.047	135.84891	0.0000000000	1525.035

Another argument to `model.table` called `model.name` is by default `TRUE` but if you set it to `FALSE` it will show the name of the R objects used to specify parameters rather than the formulas for each parameter. I find this useful when the formulas become complex and when there are several parameters in the model. Instead of using extensions like `.sex` or `.age` I use `.1`, `.2`, etc and set `model.name = FALSE` to get the results shown below.

```
do_analysis=function()
{
  # create formulas for Phi
  Phi.1=list(formula=~1)
  Phi.2=list(formula=~sex)
  Phi.3=list(formula=~sex+age)
  Phi.4=list(formula=~sex+weight)
  #create formulas for p
  p.1=list(formula=~1)
  p.2=list(formula=~time)
  # create all combinations (8 models)
  cml=create.model.list("CJS")
  # run all all 8 models and return as a list with class marklist
  results=mark.wrapper(cml,data=dp,ddl=ddl,output=FALSE,silent=TRUE)
  return(results)
}

# Now call the function to run the models and return the results
# stored into example.results
example.results=do_analysis()
# reset model names and use -2lnL
example.results$model.table=model.table(example.results,use.lnl=TRUE,model.name=FALSE)
# Show the model selection table
example.results
```

##	model	npar	AICc	DeltaAICc	weight	Neg2LnL
## 8	Phi.4.p.2	6	1393.198	0.00000	0.9648913852	1381.114
## 4	Phi.2.p.2	5	1400.498	7.29986	0.0250803711	1390.438
## 6	Phi.3.p.2	6	1402.521	9.32290	0.0091208732	1390.437
## 2	Phi.1.p.2	4	1407.137	13.93844	0.0009073705	1399.097
## 7	Phi.4.p.1	4	1515.238	122.03974	0.0000000000	1507.198
## 3	Phi.2.p.1	3	1522.122	128.92373	0.0000000000	1516.098
## 5	Phi.3.p.1	4	1523.536	130.33764	0.0000000000	1515.496
## 1	Phi.1.p.1	2	1529.047	135.84891	0.0000000000	1525.035

This is all very tidy having all of the models and the `model.table` in a single object called `example.results` here; but, you may be wondering how you retrieve results from a particular model. You use the model number listed in the `model.table` to retrieve that model. Below I use code to retrieve the model number of the best model (the first one in the table) and then show a summary of the results. Anywhere a model object can be used you can use `example.results[[#]]` where `#` is a model number.

```
# summarize the best model
best.model.number=as.numeric(row.names(example.results$model.table)[1])
summary(example.results[[best.model.number]])
```



```

## Output summary for CJS model
## Name : Phi(~sex + weight)p(~time)
##
## Npar : 6
## -2lnL: 1381.114
## AICc : 1393.198
##
## Beta
##           estimate      se      lcl      ucl
## Phi:(Intercept) -0.1928068 0.7777052 -1.7171091 1.3314956
## Phi:sexMale      1.0392923 0.3824613  0.2896681 1.7889165
## Phi:weight       0.4939090 0.1697660  0.1611676 0.8266504
## p:(Intercept)   -0.1884220 0.1497161 -0.4818656 0.1050215
## p:time3         2.5053610 0.2613115  1.9931905 3.0175316
## p:time4         1.1161492 0.1902019  0.7433535 1.4889450
##
##
## Real Parameter Phi
## Group:sexFemale.ageclassHatch-Year
##           1           2           3
## 1 0.907281 0.907281 0.907281
## 2           0.907281 0.907281
## 3           0.907281
##
## Group:sexMale.ageclassHatch-Year
##           1           2           3
## 1 0.9651144 0.9651144 0.9651144
## 2           0.9651144 0.9651144
## 3           0.9651144
##
## Group:sexFemale.ageclassAdult
##           1           2           3
## 1 0.907281 0.907281 0.907281
## 2           0.907281 0.907281
## 3           0.907281
##
## Group:sexMale.ageclassAdult
##           1           2           3
## 1 0.9651144 0.9651144 0.9651144
## 2           0.9651144 0.9651144
## 3           0.9651144
##
##
## Real Parameter p
## Group:sexFemale.ageclassHatch-Year
##           2           3           4
## 1 0.4530334 0.9102702 0.716614
## 2           0.9102702 0.716614
## 3           0.716614
##
## Group:sexMale.ageclassHatch-Year
##           2           3           4
## 1 0.4530334 0.9102702 0.716614

```

```
## 2          0.9102702 0.716614
## 3              0.716614
##
## Group:sexFemale.ageclassAdult
##          2          3          4
## 1 0.4530334 0.9102702 0.716614
## 2          0.9102702 0.716614
## 3              0.716614
##
## Group:sexMale.ageclassAdult
##          2          3          4
## 1 0.4530334 0.9102702 0.716614
## 2          0.9102702 0.716614
## 3              0.716614
```

9 Model Averaging

Unlike the example above, usually there is no clear best model and you may want to compute a weighted average of the parameters across the set of models. The model weights are the AICc derived weights shown in `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 our example, I will use `model.average` for `p` and for `Phi` I will use `covariate.predictions` in section 12 because it has `weight` as an individual covariate. If you don't specify a parameter in the call to `model.average` (e.g., `model.average(example.results)`) all of the parameters are model averaged but none of the matching design data are included because design data can vary across parameter types so there is no good way to create a single dataframe that includes the design data with the estimates. To see the design data as well, specify the `parameter` argument.

The `model.average` function is straightforward for a single type of parameter. If you only want the estimates and standard errors use the default value `vcv=FALSE`.

```
# get a dataframe with model average estimates, standard errors and related data
mavg=model.average(example.results,parameter="p")
# display the first 5 columns and the time column
mavg[,c(1:5,9)]
```

##		par.index	estimate	se	fixed	note	time
##	p gFemaleHatch-Year c1 a1 t2	25	0.4530413	0.03710062			2
##	p gFemaleHatch-Year c1 a2 t3	26	0.9102760	0.01793348			3
##	p gFemaleHatch-Year c1 a3 t4	27	0.7167026	0.02473761			4
##	p gFemaleHatch-Year c2 a1 t3	28	0.9102760	0.01793348			3
##	p gFemaleHatch-Year c2 a2 t4	29	0.7167026	0.02473761			4
##	p gFemaleHatch-Year c3 a1 t4	30	0.7167026	0.02473761			4
##	p gMaleHatch-Year c1 a1 t2	31	0.4530413	0.03710062			2
##	p gMaleHatch-Year c1 a2 t3	32	0.9102760	0.01793348			3
##	p gMaleHatch-Year c1 a3 t4	33	0.7167026	0.02473761			4
##	p gMaleHatch-Year c2 a1 t3	34	0.9102760	0.01793348			3
##	p gMaleHatch-Year c2 a2 t4	35	0.7167026	0.02473761			4
##	p gMaleHatch-Year c3 a1 t4	36	0.7167026	0.02473761			4
##	p gFemaleAdult c1 a2 t2	37	0.4530413	0.03710062			2
##	p gFemaleAdult c1 a3 t3	38	0.9102760	0.01793348			3
##	p gFemaleAdult c1 a4 t4	39	0.7167026	0.02473761			4
##	p gFemaleAdult c2 a2 t3	40	0.9102760	0.01793348			3

```

## p gFemaleAdult c2 a3 t4      41 0.7167026 0.02473761      4
## p gFemaleAdult c3 a2 t4      42 0.7167026 0.02473761      4
## p gMaleAdult c1 a2 t2        43 0.4530413 0.03710062      2
## p gMaleAdult c1 a3 t3        44 0.9102760 0.01793348      3
## p gMaleAdult c1 a4 t4        45 0.7167026 0.02473761      4
## p gMaleAdult c2 a2 t3        46 0.9102760 0.01793348      3
## p gMaleAdult c2 a3 t4        47 0.7167026 0.02473761      4
## p gMaleAdult c3 a2 t4        48 0.7167026 0.02473761      4

# display the p estimates with the standard error from top model
summary(example.results[[best.model.number]],se=TRUE)$reals$p[,1:4]

##               all.diff.index par.index estimate      se
## p gFemaleHatch-Year c1 a1 t2      25         3 0.4530334 0.0370988
## p gFemaleHatch-Year c1 a2 t3      26         4 0.9102702 0.0179357
## p gFemaleHatch-Year c1 a3 t4      27         5 0.7166140 0.0247229
## p gFemaleHatch-Year c2 a1 t3      28         4 0.9102702 0.0179357
## p gFemaleHatch-Year c2 a2 t4      29         5 0.7166140 0.0247229
## p gFemaleHatch-Year c3 a1 t4      30         5 0.7166140 0.0247229
## p gMaleHatch-Year c1 a1 t2       31         3 0.4530334 0.0370988
## p gMaleHatch-Year c1 a2 t3       32         4 0.9102702 0.0179357
## p gMaleHatch-Year c1 a3 t4       33         5 0.7166140 0.0247229
## p gMaleHatch-Year c2 a1 t3       34         4 0.9102702 0.0179357
## p gMaleHatch-Year c2 a2 t4       35         5 0.7166140 0.0247229
## p gMaleHatch-Year c3 a1 t4       36         5 0.7166140 0.0247229
## p gFemaleAdult c1 a2 t2          37         3 0.4530334 0.0370988
## p gFemaleAdult c1 a3 t3          38         4 0.9102702 0.0179357
## p gFemaleAdult c1 a4 t4          39         5 0.7166140 0.0247229
## p gFemaleAdult c2 a2 t3          40         4 0.9102702 0.0179357
## p gFemaleAdult c2 a3 t4          41         5 0.7166140 0.0247229
## p gFemaleAdult c3 a2 t4          42         5 0.7166140 0.0247229
## p gMaleAdult c1 a2 t2            43         3 0.4530334 0.0370988
## p gMaleAdult c1 a3 t3            44         4 0.9102702 0.0179357
## p gMaleAdult c1 a4 t4            45         5 0.7166140 0.0247229
## p gMaleAdult c2 a2 t3            46         4 0.9102702 0.0179357
## p gMaleAdult c2 a3 t4            47         5 0.7166140 0.0247229
## p gMaleAdult c3 a2 t4            48         5 0.7166140 0.0247229

# show structure if vcv=TRUE
str(model.average(example.results,parameter="p",vcv=TRUE))

## List of 2
## $ estimates:'data.frame': 24 obs. of 17 variables:
## ..$ par.index : int [1:24] 25 26 27 28 29 30 31 32 33 34 ...
## ..$ estimate  : num [1:24] 0.453 0.91 0.717 0.91 0.717 ...
## ..$ se        : num [1:24] 0.0371 0.0179 0.0247 0.0179 0.0247 ...
## ..$ lcl       : num [1:24] 0.382 0.868 0.666 0.868 0.666 ...
## ..$ ucl       : num [1:24] 0.526 0.94 0.763 0.94 0.763 ...
## ..$ fixed     : Factor w/ 1 level " ": 1 1 1 1 1 1 1 1 1 1 ...
## ..$ note      : Factor w/ 1 level " ": 1 1 1 1 1 1 1 1 1 1 ...
## ..$ group     : Factor w/ 4 levels "FemaleHatch-Year",...: 1 1 1 1 1 1 2 2 2 2 ...
## ..$ cohort    : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 3 1 1 1 2 ...
## ..$ age       : Factor w/ 4 levels "1","2","3","4": 1 2 3 1 2 1 1 2 3 1 ...
## ..$ time      : Factor w/ 3 levels "2","3","4": 1 2 3 2 3 3 1 2 3 2 ...
## ..$ occ.cohort: num [1:24] 1 1 1 2 2 3 1 1 1 2 ...

```

```
## ..$ Cohort      : num [1:24] 0 0 0 1 1 2 0 0 0 1 ...
## ..$ Age         : num [1:24] 1 2 3 1 2 1 1 2 3 1 ...
## ..$ Time        : num [1:24] 0 1 2 1 2 2 0 1 2 1 ...
## ..$ sex         : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 2 2 2 2 ...
## ..$ ageclass    : Factor w/ 2 levels "Hatch-Year","Adult": 1 1 1 1 1 1 1 1 1 1 ...
## $ vcv.real      : num [1:24, 1:24] 1.38e-03 2.37e-05 2.65e-05 2.37e-05 2.65e-05 ...
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:24] "25" "26" "27" "28" ...
## .. ..$ : chr [1:24] "25" "26" "27" "28" ...
```

Because the model weight for the best model is so high there is little difference between the model averaged values and the values for the best model. For the sake of consistency I need to change the label `par.index` to `model.index`. As shown above, if you want to get a variance-covariance matrix, add `vcv=TRUE` and `model.average` will return a list with names `estimates` (the dataframe containing estimates etc) and `vcv` which is the variance-covariance matrix.

The only complication is if you want to get a variance-covariance matrix of model averaged parameters across parameter types. In that case, instead of specifying the parameter name, you specify the `model.index` values (all different indices) to specify the subset of real parameters you want to see. Here I'll simply use indices from `p` as a demonstration. This same approach is used in `covariate.predictions` in section 12.

```
mavg=model.average(example.results,indices=25:26,vcv=TRUE)
mavg

## $estimates
##   par.index estimate      se      lcl      ucl
## 1         25 0.4530413 0.03710062 0.3818173 0.5262413
## 2         26 0.9102760 0.01793348 0.8683732 0.9397646
##
## $vcv.real
##           25          26
## 25 1.376456e-03 2.370519e-05
## 26 2.370519e-05 3.216096e-04
```

10 Goodness of Fit and Overdispersion

There is no single goodness of fit test for all capture-recapture models and for many models there has been no goodness of fit test developed. There is a goodness of fit test available for CJS/JS type models that is computed with program RELEASE which unfortunately only runs on Windows machines. Below is an example of `release.gof` which runs RELEASE and extracts the results.

```
release.gof(dp)

##           Chi.square df      P
## TEST2      2.1972   4 0.6995
## TEST3     15.5175  12 0.2143
## Total     17.7147  16 0.3409
```

For our example, the test fails to reject so there is no apparent lack of fit. I'd hope not since these are simulated data. But you need to know what is being tested. First of all, the model that is tested is $\text{Phi}(g \times t)p(g \times t)$ where t is time and g is group. For our example, we have 4 groups: male-female of initial ages 0 and 1Plus. This is equivalent to fitting the model $\text{Phi}(\text{sex} \times \text{ageclass} \times \text{time})p(\text{sex} \times \text{ageclass} \times \text{time})$ where `ageclass` is the initial age. 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 that are all the same age. If all of your initial release cohorts are of the same age, then Test3 (cohort differences) will not exist. As in this case, `g*t` will often provide a global model that is bigger than you would really want but that is the only option with RELEASE. See Chapter 5 of Cooch and White book on MARK for a detailed description of the RELEASE goodness of fit test and estimating over-dispersion. For another option you can look into U-CARE (also described in Chapter 5) but I believe it also only works on Windows.

There is a general method for estimating over-dispersion. One is a bootstrap approach that only works if there are no individual covariates and the other is the median chat approach that works in all cases. See chapter 5 for a description of both. Neither of these works from RMark and only runs within the MARK interface at this point so it is necessary to export the global model to the MARK interface (see section 11) and then run those procedures from the MARK interface. RMark can call RELEASE to compute Test2+Test3 chi-square tests for a CJS model to obtain an estimate \hat{c} for overdispersion, if that is appropriate.

Once you have the estimate of \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 \hat{c} 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 nor will it adjust values in the mark model objects which is why you should use `coef` and `summary` to extract results if you have adjusted \hat{c} unless you set \hat{c} prior to running the models.

11 Exporting 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 \hat{c} 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. If you have read elsewhere (like in C.21 of Appendix C of Cooch and White) about using the function `export.chdata` to export data and models into MARK interface, do **NOT** use that approach. Even though I had a warning in the help file about making sure the structure was setup the same in MARK as in RMark, errors were made and confusion and questions resulted because the results did not match when they were re-run in the MARK interface. Thus, to avoid those problems and make it much easier I created the function `export.MARK` and Gary White added the File/RMark Import menu item to the MARK interface. To export our example data and results for import into the MARK interface, use the following:

```
export.MARK(dp,project="myexample",model=example.results)

## NULL
```

If it works fine, you'll simply see NULL returned. Next, open the MARK interface and use File/RMark Import to import the results by clicking on `myexample.Rinp` and you should see something like:

Program MARK Interface - NA (C:\Users\Laake\Desktop\RMark Workshop\myexample.DBF)

File Delete Order Output Retrieve PIM Design Run Simulations Tests Adjustments Window Help

Results Browser: Live Recaptures (CJS)

Model	AICc	Delta AICc	AICc Weight	Model Likelihood	No. Par.	Deviance
{ Phi(~sex + weight)p(~time) }	1408.4155	0.0000	0.45533	1.0000	6	1396.3322
{ Phi(~sex + weight)p(~time) }	1408.4155	0.0000	0.45533	1.0000	6	1396.3322
{ Phi(~sex)p(~time) }	1413.7947	5.3792	0.03092	0.0679	5	53.1911
{ Phi(~sex)p(~time) }	1413.7947	5.3792	0.03092	0.0679	5	53.1911
{ Phi(~sex + age)p(~time) }	1415.7206	7.3051	0.01180	0.0259	6	53.0932
{ Phi(~sex + age)p(~time) }	1415.7206	7.3051	0.01180	0.0259	6	53.0932
{ Phi(~1)p(~time) }	1419.3284	10.9129	0.00194	0.0043	4	60.7446
{ Phi(~1)p(~time) }	1419.3284	10.9129	0.00194	0.0043	4	60.7446
{ Phi(~sex + weight)p(~1) }	1494.6428	86.2273	0.00000	0.0000	4	1486.6032
{ Phi(~sex + weight)p(~1) }	1494.6428	86.2273	0.00000	0.0000	4	1486.6032
{ Phi(~sex)p(~1) }	1498.7897	90.3742	0.00000	0.0000	3	142.2219
{ Phi(~sex)p(~1) }	1498.7897	90.3742	0.00000	0.0000	3	142.2219
{ Phi(~sex + age)p(~1) }	1500.5364	92.1209	0.00000	0.0000	4	141.9527
{ Phi(~sex + age)p(~1) }	1500.5364	92.1209	0.00000	0.0000	4	141.9527
{ Phi(~1)p(~1) }	1502.9702	94.5547	0.00000	0.0000	2	148.4142
{ Phi(~1)p(~1) }	1502.9702	94.5547	0.00000	0.0000	2	148.4142

Even though I said you shouldn't see the deviance vary due to use of an individual covariate in the MARK interface it does here because the models were run by creating the .inp file from RMark. If you re-run say $\text{Phi}(\sim \text{sex})\text{p}(\sim \text{time})$ by selecting that model and then using Retrieve from the menu and running it, you'll get a warning about the saturated model deviance changing. All is okay and that message is due to the model being originally run from RMark without individual covariates. Once it is re-run, the deviance will be in line with the values of models that have individual covariates in the formula.

12 Covariate Predictions

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 estimates of female survival for age 0 and 1Plus at weight values from 1 to 10.

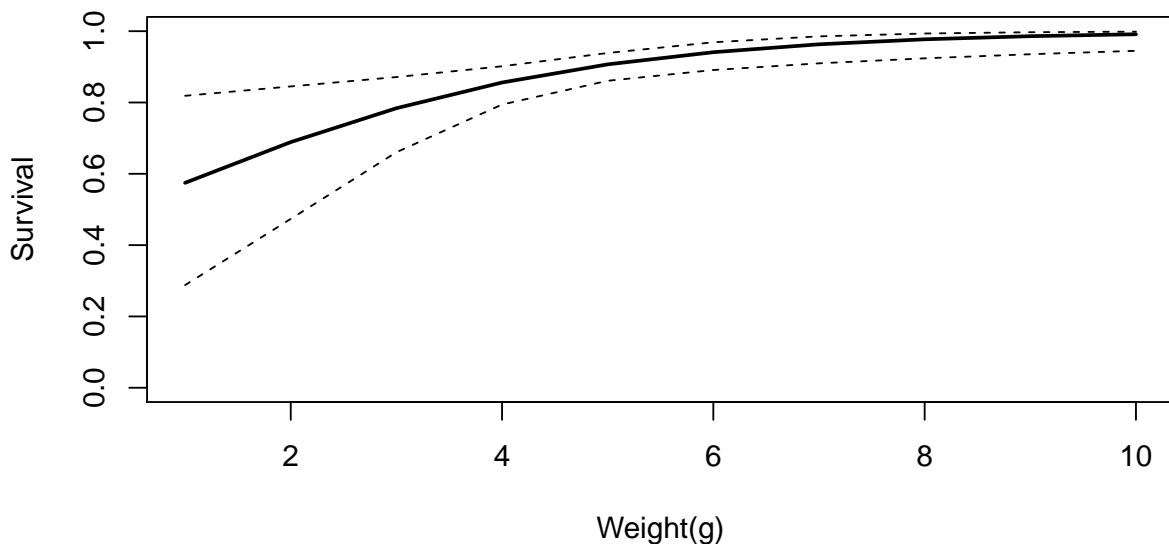
```
Phi.weight.predictions=covariate.predictions(example.results[[best.model.number]],
      data=data.frame(index=rep(1:2,each=10),weight=rep(1:10,2)))
# see results excluding first 2 columns so it fits
Phi.weight.predictions$estimates[,-(1:2)]
```

##	par.index	index	weight	estimate	se	lcl	ucl	fixed
## 1	1	1	1	0.5747119	0.150532858	0.2878134	0.8187990	
## 2	1	1	2	0.6889063	0.098546917	0.4734640	0.8450464	
## 3	1	1	3	0.7839644	0.053879661	0.6604696	0.8712945	
## 4	1	1	4	0.8560459	0.027171000	0.7942397	0.9015871	
## 5	1	1	5	0.9069321	0.019553250	0.8608888	0.9388189	
## 6	1	1	6	0.9410689	0.018917316	0.8911045	0.9689081	
## 7	1	1	7	0.9631927	0.017247855	0.9097845	0.9854872	
## 8	1	1	8	0.9772119	0.014339795	0.9238849	0.9934427	
## 9	1	1	9	0.9859693	0.011155583	0.9353443	0.9970790	
## 10	1	1	10	0.9913908	0.008290539	0.9449260	0.9987078	

```
## 11      2      2      1 0.5747119 0.150532858 0.2878134 0.8187990
## 12      2      2      2 0.6889063 0.098546917 0.4734640 0.8450464
## 13      2      2      3 0.7839644 0.053879661 0.6604696 0.8712945
## 14      2      2      4 0.8560459 0.027171000 0.7942397 0.9015871
## 15      2      2      5 0.9069321 0.019553250 0.8608888 0.9388189
## 16      2      2      6 0.9410689 0.018917316 0.8911045 0.9689081
## 17      2      2      7 0.9631927 0.017247855 0.9097845 0.9854872
## 18      2      2      8 0.9772119 0.014339795 0.9238849 0.9934427
## 19      2      2      9 0.9859693 0.011155583 0.9353443 0.9970790
## 20      2      2     10 0.9913908 0.008290539 0.9449260 0.9987078
```

The `covariate.predictions` 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 to produce a plot of the weight-Phi relationship. To keep it simple I'm using base R graphics to plot the weight-Phi relationship with point-wise confidence intervals:

```
with(Phi.weight.predictions$estimates[1:10,],{
  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)
})
```



If you use individual covariates in a model, 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 the example above. 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 our 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 6 classes (2 sexes * 3 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=apply(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 RMarkData as follows:

```
RMarkData$cohort=create.cohort(RMarkData)
summary(RMarkData)
```

##	ch	sex	weight	cohort
##	Length:600	Female:300	Min. :1.700	1:200
##	Class :character	Male :300	1st Qu.:4.317	2:200
##	Mode :character		Median :5.020	3:200
##			Mean :5.008	
##			3rd Qu.:5.652	
##			Max. :8.340	

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

```
mean.wts=with(RMarkData, tapply(weight, list(cohort,sex), mean))
mean.wts
```

##	Female	Male
## 1	5.0732	5.0677
## 2	4.9806	5.0659
## 3	4.9517	4.9112

We don't want the values for cohort 4 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 3 rows of the `mean.wts` to a vector with 6 values:

```
mean.wts=as.vector(mean.wts)
mean.wts
```

```
## [1] 5.0732 4.9806 4.9517 5.0677 5.0659 4.9112
```

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`. We do have age effects in some of the models but I'm ignoring that here so we can use a single Phi for females and a single Phi for males. By looking at `ddl$Phi` we can see that `model.index` values 1 and 7 for females and males, respectively. The following will get the predictions that we want:


```
Phi.by.wt=covariate.predictions(example.results,
                                data=data.frame(index=rep(c(1,7),each=3),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(1,7)` which alternates the female and male indices to match the order in the `mean.wts` vector. 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., `RMarkData`).

```
names(Phi.by.wt)

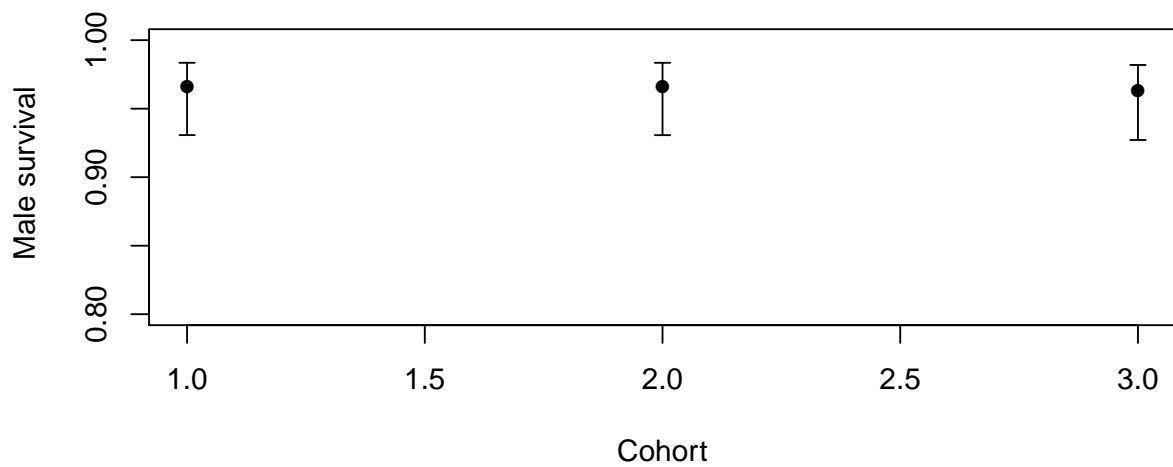
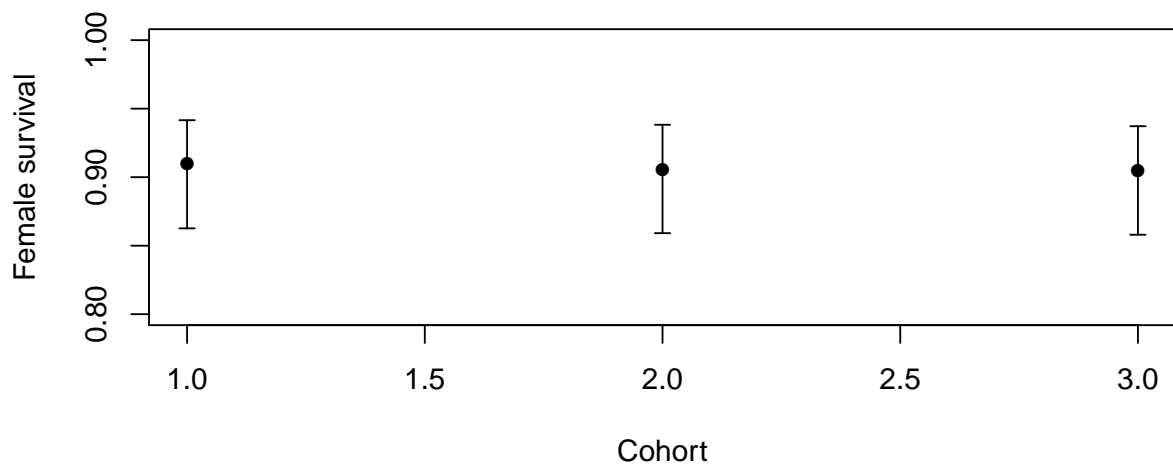
## [1] "estimates" "vcv"          "reals"

Phi.by.wt$estimates

##      vcv.index model.index par.index index weight estimate      se      lcl
## 1          1          1          1      1 5.0732 0.9095810 0.01976623 0.8626489
## 2          1          1          1      1 4.9806 0.9058961 0.01984773 0.8591433
## 3          1          1          1      1 4.9517 0.9047177 0.01988804 0.8579768
## 4          1          1          7      7 5.0677 0.9658720 0.01252871 0.9307299
## 5          1          1          7      7 5.0659 0.9658439 0.01253438 0.9306913
## 6          1          1          7      7 4.9112 0.9633385 0.01306445 0.9271454
##          ucl fixed
## 1 0.9415631
## 2 0.9382463
## 3 0.9372017
## 4 0.9835019
## 5 0.9834839
## 6 0.9819023
```

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`.

```
par(mfrow=c(2,1))
library(Hmisc)
# use ?errbar to learn more
with(Phi.by.wt$estimates,errbar(1:3,estimate[1:3],lcl[1:3],ucl[1:3],
                                ylim=c(.8,1),xlab="Cohort",ylab="Female survival"))
with(Phi.by.wt$estimates, errbar(1:3,estimate[4:6],lcl[4:6],ucl[4:6],
                                ylim=c(.8,1),xlab="Cohort",ylab=" Male survival"))
```



There is little difference in the estimates because the weights were assigned randomly. The individual estimates for each model are contained in `reals`. Below are the estimates for the top 2 models:

```
top2=as.numeric(row.names(example.results$model.table)[1:2])
Phi.by.wt$reals[top2]

## $`8`
##      estimate      se
## 1 0.9099391 0.01944449
## 2 0.9061201 0.01958717
## 3 0.9048988 0.01964210
## 4 0.9660874 0.01237676
## 5 0.9660583 0.01238322
## 6 0.9634617 0.01297241
##
```

```
## $`4`
##      estimate      se
## 1 0.8986297 0.01910736
## 2 0.8986297 0.01910736
## 3 0.8986297 0.01910736
## 4 0.9606011 0.01321040
## 5 0.9606011 0.01321040
## 6 0.9606011 0.01321040
```

In the second model, the estimates vary by sex but do not vary by cohort because weight is not in the model so the estimates are constant across cohorts. The **estimates** are averaged across models based on the model weights and the standard errors and variance-covariance matrix incorporate model uncertainty.

When you have time-varying individual covariates, you will want to specify values for each covariate. I'll use the temperature model from above as a demonstration.

```
cov.data=data.frame(temp1=5:15,temp2=5:15,temp3=5:15)
covariate.predictions(temp.model,data=cov.data,indices=1:3)$estimates[,4:10]
```

```
##      temp1 temp2 temp3 estimate      se      lcl      ucl
## 1      5      5      5 0.5147834 0.168591240 0.2203271 0.7993221
## 2      5      5      5 0.5147834 0.168591240 0.2203271 0.7993221
## 3      5      5      5 0.5147834 0.168591240 0.2203271 0.7993221
## 4      6      6      6 0.6328583 0.125005874 0.3751934 0.8318783
## 5      6      6      6 0.6328583 0.125005874 0.3751934 0.8318783
## 6      6      6      6 0.6328583 0.125005874 0.3751934 0.8318783
## 7      7      7      7 0.7368861 0.078871212 0.5578721 0.8614233
## 8      7      7      7 0.7368861 0.078871212 0.5578721 0.8614233
## 9      7      7      7 0.7368861 0.078871212 0.5578721 0.8614233
## 10     8      8      8 0.8198297 0.042719368 0.7207805 0.8891465
## 11     8      8      8 0.8198297 0.042719368 0.7207805 0.8891465
## 12     8      8      8 0.8198297 0.042719368 0.7207805 0.8891465
## 13     9      9      9 0.8808540 0.022000427 0.8305701 0.9176934
## 14     9      9      9 0.8808540 0.022000427 0.8305701 0.9176934
## 15     9      9      9 0.8808540 0.022000427 0.8305701 0.9176934
## 16    10     10     10 0.9231467 0.015278999 0.8873334 0.9482401
## 17    10     10     10 0.9231467 0.015278999 0.8873334 0.9482401
## 18    10     10     10 0.9231467 0.015278999 0.8873334 0.9482401
## 19    11     11     11 0.9512577 0.013983259 0.9153002 0.9724103
## 20    11     11     11 0.9512577 0.013983259 0.9153002 0.9724103
## 21    11     11     11 0.9512577 0.013983259 0.9153002 0.9724103
## 22    12     12     12 0.9694269 0.012492468 0.9327959 0.9863829
## 23    12     12     12 0.9694269 0.012492468 0.9327959 0.9863829
## 24    12     12     12 0.9694269 0.012492468 0.9327959 0.9863829
## 25    13     13     13 0.9809590 0.010340934 0.9456703 0.9934846
## 26    13     13     13 0.9809590 0.010340934 0.9456703 0.9934846
## 27    13     13     13 0.9809590 0.010340934 0.9456703 0.9934846
## 28    14     14     14 0.9881941 0.008061585 0.9557616 0.9969259
## 29    14     14     14 0.9881941 0.008061585 0.9557616 0.9969259
## 30    14     14     14 0.9881941 0.008061585 0.9557616 0.9969259
## 31    15     15     15 0.9927005 0.006021749 0.9638696 0.9985597
## 32    15     15     15 0.9927005 0.006021749 0.9638696 0.9985597
## 33    15     15     15 0.9927005 0.006021749 0.9638696 0.9985597
```

In this case, the individual covariate values are the same for all individuals so I could have used just used **index=1** and **temp1** values, but when you have values that vary across individuals, you could have an

interaction of time and the individual covariate and you would need to use the more general formulation above.

13 Numerical 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(example.results$model.table)

## [1] "Phi"      "p"        "model"    "npar"     "AICc"     "DeltaAICc"
## [7] "weight"   "Neg2LnL"
```

The contents of Phi and p are the model object name for that model parameter because we set `model.name=FALSE`. 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 we can get all the AICc values in a matrix form:

```
with(example.results$model.table, tapply(AICc, list(Phi,p), unique))

##              ~1      ~time
## ~1          1529.047 1407.137
## ~sex        1522.122 1400.498
## ~sex + age  1523.536 1402.521
## ~sex + weight 1515.238 1393.198
```

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

```
with(example.results$model.table, tapply(Neg2LnL, list(Phi,p), unique))

##              ~1      ~time
## ~1          1525.035 1399.097
## ~sex        1516.098 1390.438
## ~sex + age  1515.496 1390.437
## ~sex + weight 1507.198 1381.114
```

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 to `mark` or `mark.wrapper`, you can specify initial values for the beta parameters 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:

```

# fit initial constant model
null=mark(dp,ddl,output=FALSE)
summary(null)$beta

##              estimate          se          lcl          ucl
## Phi:(Intercept) 2.9193644 0.2651801 2.3996113 3.439118
## p:(Intercept)   0.9221261 0.0898523 0.7460157 1.098237

# use null as initial for starting values
model=mark(dp,ddl,model.parameters=list(Phi=list(formula=~sex+age+weight)),
           initial=null,output=FALSE)
summary(model)$beta

##              estimate          se          lcl          ucl
## Phi:(Intercept) -0.9696015 1.0567487 -3.0408290 1.101626
## Phi:sexMale      1.5169462 0.6229587  0.2959470 2.737945
## Phi:age1Plus     0.5008667 0.4872240 -0.4540923 1.455826
## Phi:weight       0.6312871 0.2268296  0.1867010 1.075873
## p:(Intercept)   0.9228751 0.0885596  0.7492984 1.096452

```

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 the example above, the only matching values will be the intercepts. For any beta that does not 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.

14 Singularity, Overfitting and Parameter Counting

“Singular” parameters are parameters that are either 1) confounded with another parameter in the model, or 2) represent a data category (factor level) that does not have any data, or 3) from a “duplicated” DM column, or 4) 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.

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.

It is important to recognize that RMark assumes that all of your beta parameters (number of columns for your DM) 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`, `mark.wrapper` or `model.table` functions). 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 have 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 $\Phi(t)p(t)$, then only the product of the $\Phi \cdot p$ is estimable for the last occasion. For example, with our example data there are 6 beta parameters but only 5 parameters are estimable. The help files in MARK will help you determine what your parameter count should be with specific types of models.

```
model=mark(dp,ddl,model.parameters=list(Phi=list(formula=~time),p=list(formula=~time)))

##
## Note: only 5 parameters counted of 6 specified parameters
##
## AICc and parameter count have been adjusted upward
##
## Output summary for CJS model
## Name : Phi(~time)p(~time)
##
## Npar : 6 (unadjusted=5)
## -2lnL: 1398.755
## AICc : 1410.839 (unadjusted=1408.8152)
##
## Beta
##
## estimate se lcl ucl
## Phi:(Intercept) 2.4277478 0.3545758 1.7327792 3.1227165
## Phi:time2 0.3350811 0.5588405 -0.7602464 1.4304086
## Phi:time3 -0.9202749 466.3977700 -915.0599200 913.2193700
## p:(Intercept) -0.1721891 0.1524259 -0.4709438 0.1265657
## p:time3 2.4686952 0.2695996 1.9402799 2.9971105
## p:time4 1.6504690 455.4124500 -890.9579600 894.2589000
##
##
## Real Parameter Phi
## Group:sexFemale.ageclassHatch-Year
## 1 2 3
## 1 0.9189189 0.9406338 0.8186864
## 2 0.9406338 0.8186864
## 3 0.8186864
##
## Group:sexMale.ageclassHatch-Year
## 1 2 3
## 1 0.9189189 0.9406338 0.8186864
## 2 0.9406338 0.8186864
## 3 0.8186864
##
```

```
## Group:sexFemale.ageclassAdult
##           1           2           3
## 1 0.9189189 0.9406338 0.8186864
## 2           0.9406338 0.8186864
## 3           0.8186864
##
## Group:sexMale.ageclassAdult
##           1           2           3
## 1 0.9189189 0.9406338 0.8186864
## 2           0.9406338 0.8186864
## 3           0.8186864
##
##
## Real Parameter p
## Group:sexFemale.ageclassHatch-Year
##           2           3           4
## 1 0.4570588 0.9085873 0.8143126
## 2           0.9085873 0.8143126
## 3           0.8143126
##
## Group:sexMale.ageclassHatch-Year
##           2           3           4
## 1 0.4570588 0.9085873 0.8143126
## 2           0.9085873 0.8143126
## 3           0.8143126
##
## Group:sexFemale.ageclassAdult
##           2           3           4
## 1 0.4570588 0.9085873 0.8143126
## 2           0.9085873 0.8143126
## 3           0.8143126
##
## Group:sexMale.ageclassAdult
##           2           3           4
## 1 0.4570588 0.9085873 0.8143126
## 2           0.9085873 0.8143126
## 3           0.8143126
```

Parameters that are at a boundary are easy to identify because they usually have standard errors that are 0 or 10 or 100 times or more greater than the beta estimate. If all of the standard errors for the beta estimates for a particular parameter (e.g., Φ) 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. As an example, I'll miss classify the times in our example data to include a time that is not in the design data which will incorrectly end up being the intercept in the model:

```
summary(ddl$p$time)

##  2  3  4
##  4  8 12
```

```

ddl$p$timebin=cut(ddl$p$Time,c(-2,-1,0,1,2))
levels(ddl$p$timebin)

## [1] "(-2,-1]" "(-1,0]" "(0,1]" "(1,2]"

model=mark(dp,ddl, model.parameters=list(p=list(formula=~timebin)))

##
## 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: 1399.097
## AICc : 1409.156 (unadjusted=1407.1365)
##
## Beta
##
## estimate se lcl ucl
## Phi:(Intercept) 2.6211169 0.1953676 2.238196 3.004037
## p:(Intercept) 0.8132267 172.9328000 -338.135060 339.761520
## p:timebin(-1,0] -1.0020489 172.9328500 -339.950440 337.946350
## p:timebin(0,1] 1.5043835 172.9329000 -337.444110 340.452870
## p:timebin(1,2] 0.1116135 172.9327800 -338.836650 339.059870
##
##
## Real Parameter Phi
## Group:sexFemale.ageclassHatch-Year
## 1 2 3
## 1 0.9322083 0.9322083 0.9322083
## 2 0.9322083 0.9322083
## 3 0.9322083
##
## Group:sexMale.ageclassHatch-Year
## 1 2 3
## 1 0.9322083 0.9322083 0.9322083
## 2 0.9322083 0.9322083
## 3 0.9322083
##
## Group:sexFemale.ageclassAdult
## 1 2 3
## 1 0.9322083 0.9322083 0.9322083
## 2 0.9322083 0.9322083
## 3 0.9322083
##
## Group:sexMale.ageclassAdult
## 1 2 3
## 1 0.9322083 0.9322083 0.9322083
## 2 0.9322083 0.9322083
## 3 0.9322083
##
##

```



```
## Real Parameter p
## Group:sexFemale.ageclassHatch-Year
##           2           3           4
## 1 0.4529342 0.910325 0.7160273
## 2           0.910325 0.7160273
## 3           0.7160273
##
## Group:sexMale.ageclassHatch-Year
##           2           3           4
## 1 0.4529342 0.910325 0.7160273
## 2           0.910325 0.7160273
## 3           0.7160273
##
## Group:sexFemale.ageclassAdult
##           2           3           4
## 1 0.4529342 0.910325 0.7160273
## 2           0.910325 0.7160273
## 3           0.7160273
##
## Group:sexMale.ageclassAdult
##           2           3           4
## 1 0.4529342 0.910325 0.7160273
## 2           0.910325 0.7160273
## 3           0.7160273
```

In the summary 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. All of the beta parameters for p have very large standard errors and the reason is that the intercept level (-2,-1] does not exist in the design data. Now if we were to relevel the timebin factor variable so (-1,0] is the intercept then we get the following summary:

```
ddl$p$timebin=relevel(ddl$p$timebin,"(-1,0]")
model=mark(dp,ddl, model.parameters=list(p=list(formula=~timebin)),output=FALSE)
summary(model)$beta
```

	estimate	se	lcl	ucl
## Phi:(Intercept)	2.6211170	0.1953676	2.2381965	3.0040375
## p:(Intercept)	-0.1888222	0.1497076	-0.4822490	0.1046046
## p:timebin(0,1]	2.5064324	0.2614224	1.9940444	3.0188204
## p:timebin(1,2]	1.1136624	0.1909280	0.7394435	1.4878813

Hmm! There is no longer an extra parameter. 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.

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 DM that you are creating with any formula.

15 Miscellaneous Topics

15.1 Cleanup

If you have been creating models and not saving them or deleting them afterward, 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 are not 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.

15.2 Profile intervals

If parameters are estimable but at a boundary, the estimate of precision (standard error) is usually unreliable but 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. If you have individual covariates in the formula, then you can specify the values used to estimate the real parameters and profile intervals by specifying the argument `icvalues` to `mark` or `mark.wrapper`. The argument `icvalues` is a numeric vector and the order of the values should match the order of the individual covariates in the formula.

```
# compute real parameters at average weight value using hessian
model=mark(dp,ddl,model.parameters=list(Phi=list(formula=~sex+age)),
           output=FALSE)
summary(model,se=TRUE)$reals$Phi[1:12,c(3,5:6)]

##
##          estimate      lcl      ucl
## Phi gFemaleHatch-Year c1 a0 t1 0.8894703 0.7875804 0.9458477
## Phi gFemaleHatch-Year c1 a1 t2 0.9229442 0.8666117 0.9566758
## Phi gFemaleHatch-Year c1 a2 t3 0.9229442 0.8666117 0.9566758
## Phi gFemaleHatch-Year c2 a0 t2 0.8894703 0.7875804 0.9458477
## Phi gFemaleHatch-Year c2 a1 t3 0.9229442 0.8666117 0.9566758
## Phi gFemaleHatch-Year c3 a0 t3 0.8894703 0.7875804 0.9458477
## Phi gMaleHatch-Year c1 a0 t1 0.9707637 0.9065100 0.9912819
## Phi gMaleHatch-Year c1 a1 t2 0.9801669 0.9378193 0.9938628
## Phi gMaleHatch-Year c1 a2 t3 0.9801669 0.9378193 0.9938628
## Phi gMaleHatch-Year c2 a0 t2 0.9707637 0.9065100 0.9912819
## Phi gMaleHatch-Year c2 a1 t3 0.9801669 0.9378193 0.9938628
## Phi gMaleHatch-Year c3 a0 t3 0.9707637 0.9065100 0.9912819

# compute real parameters at average weight value using profile CI
model=mark(dp,ddl,model.parameters=list(Phi=list(formula=~sex+age)),
           output=FALSE,profile.int=TRUE)
summary(model,se=TRUE)$reals$Phi[1:12,c(3,5:6)]

##
##          estimate      lcl      ucl
## Phi gFemaleHatch-Year c1 a0 t1 0.8894703 0.8062137 0.9584867
## Phi gFemaleHatch-Year c1 a1 t2 0.9229443 0.8761933 0.9626423
## Phi gFemaleHatch-Year c1 a2 t3 0.9229443 0.8761933 0.9626423
## Phi gFemaleHatch-Year c2 a0 t2 0.8894703 0.8062137 0.9584867
## Phi gFemaleHatch-Year c2 a1 t3 0.9229443 0.8761933 0.9626423
## Phi gFemaleHatch-Year c3 a0 t3 0.8894703 0.8062137 0.9584867
```

```
## Phi gMaleHatch-Year c1 a0 t1 0.9707637 0.9278000 0.9983819
## Phi gMaleHatch-Year c1 a1 t2 0.9801669 0.9508663 0.9986040
## Phi gMaleHatch-Year c1 a2 t3 0.9801669 0.9508663 0.9986040
## Phi gMaleHatch-Year c2 a0 t2 0.9707637 0.9278000 0.9983819
## Phi gMaleHatch-Year c2 a1 t3 0.9801669 0.9508663 0.9986040
## Phi gMaleHatch-Year c3 a0 t3 0.9707637 0.9278000 0.9983819
```

15.3 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 error²) 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.

15.4 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`. When there are multiple types of derived parameters they are in a named list. For example below I use the dipper data in RMark with the POPAN model to show the structure of the derived parameters and the variance-covariance matrices which are computed separately for each type of derived parameter:

```
data(dipper)
model=mark(dipper,model="POPAN",output=FALSE)
str(model$results$derived)

## List of 4
## $ B* Gross Births      :'data.frame': 6 obs. of 4 variables:
## ..$ estimate: num [1:6] 62.6 62.6 62.6 62.6 62.6 ...
## ..$ se       : num [1:6] 2.04 2.04 2.04 2.04 2.04 ...
## ..$ lcl      : num [1:6] 58.7 58.7 58.7 58.7 58.7 ...
## ..$ ucl      : num [1:6] 66.7 66.7 66.7 66.7 66.7 ...
## $ B Net Births        :'data.frame': 6 obs. of 4 variables:
## ..$ estimate: num [1:6] 47.5 47.5 47.5 47.5 47.5 ...
## ..$ se       : num [1:6] 1.27 1.27 1.27 1.27 1.27 ...
## ..$ lcl      : num [1:6] 45.1 45.1 45.1 45.1 45.1 ...
```

```
## ..$ ucl      : num [1:6] 50 50 50 50 50 ...
## $ N Population Size      : 'data.frame': 7 obs. of  4 variables:
## ..$ estimate: num [1:7] 24.3 61.1 81.6 93.1 99.6 ...
## ..$ se       : num [1:7] 5.06 2.53 2.69 3.85 4.85 ...
## ..$ lcl      : num [1:7] 16.3 56.3 76.5 85.9 90.5 ...
## ..$ ucl      : num [1:7] 36.4 66.2 87.1 101 109.5 ...
## $ Gross N* Population Size: 'data.frame': 1 obs. of  4 variables:
## ..$ estimate: num 400
## ..$ se       : num 10.9
## ..$ lcl      : num 379
## ..$ ucl      : num 422

str(model$results$derived.vcv)

## List of 4
## $ B* Gross Births      : num [1:6, 1:6] 4.15 4.15 4.15 4.15 4.15 ...
## $ B Net Births         : num [1:6, 1:6] 1.61 1.61 1.61 1.61 1.61 ...
## $ N Population Size    : num [1:7, 1:7] 25.61 11.03 3.07 -1.28 -3.65 ...
## $ Gross N* Population Size: num [1, 1] 118
```

If you want to model average derived parameters, you'll need to construct lists of the estimates and variance-covariance matrices and use `model.average.list`. See `?model.average.list` for more details.

15.5 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 DM 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 use the RMark functions. Obviously you can't extract results directly from the model because it only contains the character string specifying the file that contains all of the data. 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 and you could access the results directly from it.

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.

15.6 Variance components

You can estimate variance components by exporting the model to MARK and choosing that option from the output model or you can use the function `var.components` or `var.components.reml` in RMark. See `?var.components` for a description. If you use `var.components` the calculations are done in R and not with MARK but should match the values computed from MARK. The function `var.components.reml` is more general and has a number of advantages. See `?var.components.reml` for more information.

Table 1: Models with shared parameter pairings. First parameter listed is the dominant parameter.

	Model	Parameter.pair
1	Barker	F/FPPrime
2	Closed	p/c
3	FullHet	p/c
4	Huggins	p/c
5	HugFullHet	p/c
6	MSOccupancy	p1/p2
7	RDOccupEG	Epsilon/Gamma
8	RDOccupHetEG	Epsilon/Gamma
9	Robust	GammaDoublePrime/GammaPrime
10	Robust	p/c
11	RDHet	GammaDoublePrime/GammaPrime
12	RDHFHet	GammaDoublePrime/GammaPrime
13	RDHFHet	p/c
14	RDFullHet	GammaDoublePrime/GammaPrime
15	RDFullHet	p/c
16	RDHuggins	GammaDoublePrime/GammaPrime
17	RDHuggins	p/c
18	RDHHet	GammaDoublePrime/GammaPrime
19	CRDMS	p/c
20	PoissonMR	GammaDoublePrime/GammaPrime
21	HCRDMS	p/c
22	FHetRDMS	p/c
23	HFHetRDMS	p/c
24	CRDMSOHug	p/c
25	CRDMSOFHet	p/c
26	2SpecOccup	PsiA/PsiB
27	2SpecOccup	pA/pB
28	2SpecOccup	rAb/raB
29	2SpecConOccup	PsiBA/PsiBa
30	2SpecConOccup	pA/pB
31	2SpecConOccup	rBA/rBa
32	RDPdGClosed	p/c
33	RDPdGHuggins	p/c
34	RDPdGFullHet	p/c
35	RDPdGHugFullHet	p/c
36	RDPdLClosed	p/c
37	RDPdLHuggins	p/c
38	RDPdLFullHet	p/c
39	RDPdLHugFullHet	p/c
40	RDPdfClosed	p/c
41	RDPdfHuggins	p/c
42	RDPdfFullHet	p/c
43	RDPdfHugFullHet	p/c
44	RDBarker	aDoublePrime/aPrime
45	RDBarker	p/c
46	RDBarkHet	aDoublePrime/aPrime
47	RDBarkFHet	aDoublePrime/aPrime
48	RDBarkFHet	p/c
49	RDBarkHug	aDoublePrime/aPrime
50	RDBarkHug	p/c
51	RDBarkHHet	aDoublePrime/aPrime
52	RDBarkHFHet	aDoublePrime/aPrime
53	RDBarkHFHet	p/c