APPENDIX C

RMark – an alternative approach to building linear models in MARK

Jeff Laake
Alaska Fisheries Science Center
National Marine Fisheries Service
Seattle, Washington, USA

Eric Rexstad
Research Unit for Wildlife Population Assessment
CREEM - University of St. Andrews
St. Andrews, Scotland

For most of the examples presented in the **MARK** book, the construction of the design matrix (DM) using the 'graphical DM template' is relatively straightforward. Moreover, by 'forcing' you to confront the actual structure of the design matrix, the relationship between linear models, covariates, even fundamental statistical entities like 'degrees of freedom' may actually make more sense than they did before.

However, quite often in real-world situations, where the size of design matrices can get very large, very quickly, it is often cumbersome to build design matrices in this fashion. Further, your chances of making a mistake while building the design matrix increase in rough proportion to the size of the design matrix - compounded when the models contain significant ultrastructure. Also, if either the number of occasions or group structure changes, PIMs and DMs must be changed and this means rebuilding each model in **MARK**. Thus, automated model development is almost a necessity for researchers that monitor populations over time and are continually adding sampling occasions.

This appendix describes an alternate interface that can be used in place of **MARK**'s graphical interface to describe and run models in terms of formula (e.g., `Phi~sex+age+time`). The interface constructs the necessary PIMS and design matrices which automates model development. The interface creates the **MARK** input file, initiates `mark.exe` and then extracts the results from the output files. All of the computation for parameter estimation is done with **MARK** (`mark.exe`). This alternative interface is a package that has been written in **R**, a freely available statistical programming environment.

Thus the package was named **RMark**. This appendix provides an introduction and description of the **RMark** interface to help you get started. The appendix does not document every function and every function argument in the package because that reference material is provided in the help file documentation that accompanies the package as described below. Instead, analyses of the dipper and swift datasets and other examples are repeated here using **RMark** to demonstrate this 'formula based' approach to specifying models.

In addition to automating model development, **RMark** has the following advantages:

1. labels for real (reconstituted) and β parameters are automatically added for ease of interpretation
2. scripts can be written to run an entire analysis and the script can be documented
3. covariate-specific real parameter estimates can be computed within **R** without re-running the analysis
4. the **R** environment is available for plotting and further computation on the results.

Examples of these advantages are given throughout this appendix.

However, there are a number of disadvantages in comparison to the existing **MARK** graphical interface. First and foremost, you need to have a rudimentary knowledge of **R** before using **RMark**. There is no getting around it and while it could be viewed as a disadvantage for **RMark**, it can also be viewed as an advantage because **R** is a very powerful statistical programming environment that can be useful for many different analysis tasks and **RMark** may be the push you need to start using **R** for all of your analyses. To help you learn **R**, we provide a very brief **R** primer at the end of this appendix, but we suggest that you also take advantage of the **R** tutorial material on the web and in various books. Even if you have a reasonable grasp of **R**, it may be useful to review the tutorial to understand how lists provide useful structures for working with models in **RMark**.

At present, another disadvantage is that the **RMark** interface does not replicate every aspect of the **MARK** interface. In particular, not every model in **MARK** is supported by **RMark**. For a complete list, refer to the **`MarkModels.pdf`** file that is installed in the **RMark** subdirectory (from within **MARK**, see also '**`Help | Data Types`**'). In addition, features such as the median $\hat{c}$ goodness-of-fit testing and random effects are not available at present. A solution is to export the model runs from **RMark** into the **MARK** interface (discussed later in this appendix).

Another subtle difference with the **MARK** interface is that all models constructed in the **RMark** interface are developed via a design matrix approach rather than coding the model structure via parameter index matrices (PIMS). The title for this appendix was chosen to reflect this aspect of the **RMark** interface. However, as of version 1.7.6 RMark can create models with an identity design matrix and you can now use the sin link as long as the formula specifies a model that can be represented by a identity matrix. Obviously, you cannot use covariates with the sin link. See section C.10 for more explanation. Even though **RMark** constructs the design matrix for you, you still need to understand the concepts described in the book about design matrices and counting parameters. Having the description of **RMark** in an appendix to this book is intentional and appropriate because initially it is best to learn to use the standard interface so that you understand what **RMark** is doing.

In manuscripts, cite this appendix for **RMark** and in describing it make sure to say something like "we used the **R** (R Development Core Team 2007) package **RMark** (Laake 2013) to construct models for program **MARK** (White and Burnham 1999)." Use **`citation("RMark")`** in **R** to get the proper citation for **R**.*

If you have no experience with **R** we highly recommend that you start by reading C.1 and C.24 and either a good introductory text on **R** or the online material on the **R** home page (which is found at **`http://www.r-project.org/`**). Once you become moderately comfortable with **R**, read sections C.2-C.12 and follow along with the examples. After reading section C.12 you should be able to import your own data and work through the more advanced sections in C.13-C.16. Specific examples of models beyond CJS are given in section C.17-C.20 and we expect to add more sections like this in future revisions. If

* Laake, J. L. (2013). RMark: An R Interface for Analysis of Capture-Recapture Data with MARK. AFSC Processed Rep 2013-01, 25p. Alaska Fish. Sci. Cent., NOAA, Natl. Mar. Fish. Serv., 7600 Sand Point Way NE, Seattle WA 98115.

you want to know how to export **RMark** models to use features of the **MARK** interface see section C.21. Examples of using **R** for further computation on results like creating delta method variances are described in C.22. If you encounter errors or problems, see C.23 for a list of common errors and suggested solutions.

Some of the examples displayed here will only work with version 1.7.3 of **RMark** or later and the December, 2007 version or later of `mark.exe`.

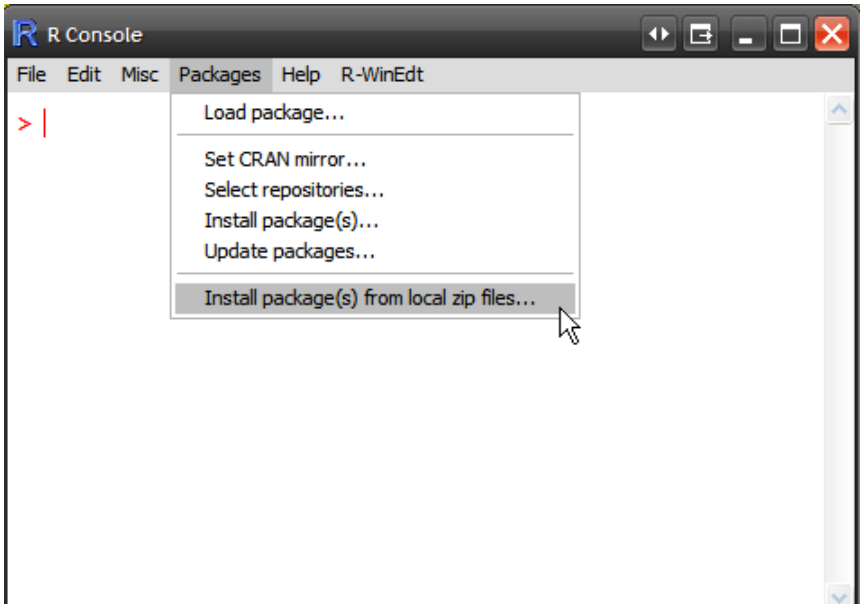
C.1. RMark installation and first steps

There are a number of tasks that you need to accomplish prior to using **RMark**. Since you are reading this appendix, chances are good that you will have already installed **MARK** on your computer. If not, refer to the Foreword of this book.

If you haven't already done so, you must install **R** from the **R** Project website

`http://cran.r-project.org/`

Select 'Windows95 or later', then select base and finally select `r-v.v.v-win32.exe` (where `v.v.v` is the current version (e.g., `R-2.6.1-win32.exe`)). Select run and then follow the directions and choose the default setup by clicking on "next" at each prompt. Download and save the **RMark** package (`RMark.zip`) from `www.phidot.org`. Start **R** from either the desktop icon or from the Start/All Programs list. From within **R**, select `Packages` from the menu and then choose `Install package(s) from local zip` at the bottom of the menu list:



Doing so will show a "select files" window. Navigate to the location where you saved the `RMark.zip` and select the zip file. This will load the package into `c:\Program Files\R\R-v.v.v\library`, where `v.v.v` represents the current version of **R** that you are using (e.g., 2.6.1). Note that **R** installs each version into separate sub-directories of `c:\Program Files\R`. Any updates for **RMark** can be installed as described above over previous versions. If you update **R** versions, you need to repeat the **RMark** installation.

You only need to install **RMark** once but to use **RMark** you will need to issue the command `library(RMark)` in **R** to attach the package every time you start **R**. **R** should respond by displaying the version number that you have installed (e.g. 1.7.3 as shown below) and it will give details about the build date and time and the version of **R** that was used to build the package:

```
This is RMark 1.7.3 Built: R 2.6.1; i386-pc-mingw32; 2007-12-20 09:57:44; windows
```

Most of the time the version of **R** that you are using can be newer than the **R** version used to build **RMark**. However, there are exceptions and if you are having problems with error messages that you cannot resolve, check that the version numbers agree. To avoid manually entering the command each time you initiate **R**, you have a couple of options. You can edit and enter the `library(RMark)` command into the file named "`RProfile.site`" with any text editor. It is located in the directory `C:\Program Files\R\R-v.v.v\etc\` where `v.v.v` represents the **R** version. If you add the library(RMark) command to rprofile.site, the **RMark** package will be loaded anytime you start **R**. For additional material on `RProfile.site` see C.24.

Alternatively, you can write a function `.First=function()library(RMark)` and save it into any `.Rdata` workspace from which you will do **MARK** analyses. The `.First()` function is run anytime that particular `.Rdata` workspace is opened. See section C.13 and **R** documentation for more on writing functions.

If you did not select the default location for **MARK** in the installation process (C:\Program Files\Mark) where **RMark** will expect it, then you need to set a variable `MarkPath` to point to the location of the `mark.exe` file. This is best to do in either the `RProfile.site` file or `.First` function. As an example, if you installed **MARK** to `d:\myfiles\mymark`, then in `RProfile.site`, then add the command `MarkPath="d:/myfiles/mymark/"` or `MarkPath="d:\\myfiles\\mymark\\"` (note: Windows uses a backslash ("\") to separate sub-directories in a path but in **R** they are either represented by either a double backslash ("\\") or the simpler single forward slash ("/")). The default value is `MarkPath="C:/Program Files/Mark/"`. Another useful variable that can be set is `MarkViewer`. By default this is set to `notepad.exe` but you can set it to any program like `wordpad.exe` or any text editor you prefer. You need to specify the full directory specification and program name unless the directory is in the `PATH` environment variable.

MARK creates several files when it runs a model (`.inp`, `.out`, `.vcv`, `.res`) and **RMark** retains and uses these four files in the directory where they were created. Everything else **RMark** creates is contained in the `.Rdata` file which is the **R** workspace. Thus, it is best to create a sub-directory for each set of data you are going to analyze with **RMark**. After you have created the sub-directory copy an empty `.Rdata` file into the new sub-directory and then you can initiate an **R** session by simply double-clicking the `.Rdata` file and any files **RMark** creates will be contained within the subdirectory. It is typically best to start with an empty .Rdata workspace. After a fresh install of **R**, the file `C:\Program Files\R\R-v.v.v\.Rdata` is empty and can be copied to start with an empty workspace. Or all of the workspace contents can be deleted using the command `rm(list=ls(all=TRUE))` or use '`remove all objects`' under the '`Misc`' item in the **R** menu. This is particularly important if you want to mimic some of the examples described in this appendix.

Beyond this appendix, there is an extensive amount of documentation written for each function contained in **RMark**. You can see this documentation using `?mark` within **R** after `library(RMark)` or to see the entire help file double-click on the file

```
C:\Program Files\R\R-v.v.v\library\RMark \chtml\RMark.chm
```

where `v.v.v` is the **R** version that you are using. From there you can view or print the entire contents (currently 144 pages).

C.2. A simple example (return of the dippers)

Let's start with a very simple example to explain some of the basic aspects of using **RMark**. Create an empty directory and copy a **.Rdata** file to it. Double click the **.Rdata** file to initiate **R** with that workspace. If there are any objects in the workspace (use **ls()** to see the contents) remove any objects the '**remove all objects**' under the '**Misc**' menu item. Type **library(Rmark)** to attach the package if you have not setup **R** such that the **RMark** package is always attached.

For your first example, we will use the well-known dipper data set that accompanies **MARK** (the dipper data, and all of the other example data files referred to in the **MARK** book and this appendix are found at `http://www.phidot.org/software/mark/docs/book/`. In the drop-down menu '**Book chapters & data files**', select '**Example data files**'). In fact, the dipper data set and a number of others are already contained in the **RMark** package and they can be accessed with the **data** function which extracts the dataframe from the library and puts a copy into your workspace. In addition, with each example set of data, there is some example code for **RMark** to demonstrate use of that particular model. You can run the example code by typing **example(dipper)**, but for this tutorial we will take a simple example and enter each command. If you type data(dipper) and then type **ls()**, it should only show **dipper** as the contents of your workspace.

```
> data(dipper)
> ls()
   [1] "dipper"
```

(Note: The object **dipper** is a *dataframe* which is equivalent to a table in MS-ACCESS). Let's get a summary of **dipper** and display the first 5 records to see that it is in the correct format for **RMark**:

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

> dipper[1:5,]
     ch    sex
 1 0000001 Female
 2 0000001 Female
 3 0000001 Female
 4 0000001 Female
 5 0000001 Female
```

From the above you see that **dipper** has a field named "**ch**" which is a character string containing the capture (encounter) history and it has a field called "**sex**" which is a factor variable. We can tell that "**sex**" is a factor variable because summary shows the frequency of the levels of the factor variable. If it was a numeric variable, summary would show the min, mean, max etc.

We can run a very simple analysis with the **mark** function and assign it to the object "**myexample**" as follows:

```
> myexample=mark(dipper)
```

The output on the screen will be:

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

Npar :  2
-2lnL:  666.8377
AICc :  670.866

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

Real Parameter Phi

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

Real Parameter p

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

So what happened and what did that do? First of all let's dissect the command. The piece of code **`mark(dipper)`** called the function **`mark`** with the data file **`dipper`** and used it for the value of its first argument which is called **`data`**. The equal sign (or `<-` can be used) was used to assign the result of the **`mark`** function to the object **`myexample`** which is stored in the **`.Rdata`** workspace (although only in memory until the workspace is saved to disk).

So what actually happened inside of the **`mark`** function? It constructed and ran an analysis using the dataframe **`dipper`** and the default values for the function arguments **`model("CJS")`** and **`model.parameters`** which for this model is to construct **`Phi(.)p(.)`** (i.e., $\{\varphi_. \; p_.\}$) in **MARK** notation and **`Phi(~1)p(~1)`** in **R** notation. It also used the default values for other function arguments such as **`time.intervals`** and assumed that there was no group structure for the analysis. Numerous steps were involved but all you need now is the abbreviated version.

The function **`mark`** examined the capture history (ch) to determine the number of occasions, developed all the necessary structure that it needed, created an **`.inp`** file for **MARK**, ran **`mark.exe`** in the background and extracted relevant parts of the **MARK** output files that it needed to create a list of results. If you use the **R** function **`list.files()`** you'll see that your directory now contains 4 more files which are the

input and the 3 output files from **mark.exe** and each with the prefix **mark001**.

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

The file **mark001.inp** is the file that would be equivalent to what you would see if you used "**Save Structure**" rather than directly running the model in the **MARK** interface. **mark001.out** is the text output file from **MARK** with all the results. **mark001.res** is the file of residuals (not currently used by **RMark**) and **mark001.vcv** is a binary file containing the variance-covariance matrices and parameter estimates. All of these files are "linked" to the result object in **R** by the base filename. In this case **myexample** is linked to "**mark001**". Files are numbered sequentially for each analysis with the first available number, but more on that later.

The results from **MARK** were put into the object **myexample** which is a list. If you don't understand the concept of a list in **R**, refer to the **R** tutorial in section C.24. The list created by **RMark** is a slightly special list because it has been assigned to a *class* which means that **R** will treat it differently based on its class. The differential treatment occurs when generic functions like **print** and **summary** are called with the object. You can see the class of an object with the **class** function as follows:

```
> class(myexample)
 [1] "mark" "CJS"
```

It has 2 classes with the first being **mark** and the second being the type of mark-recapture model which is "CJS" (Cormack-Jolly-Seber) by default. You need to know this only to understand that when you use functions like **print** and **summary** that **R** actually calls print.mark and **summary.mark**. When **MARK** was finished with the analysis it called **summary** (**summary.mark**) which created the output on the screen. You can see the output again on the screen by simply typing **summary(myexample)**. If you want to save those results to a file, you can use cut and paste to the clipboard or you can use the **sink** function to save any screen output to a file. Use **sink(myfilename)** before issuing the command that generates the output and use any valid file specification in place of **myfilename**. To restore output to the screen use **sink()**.

Let's discuss the summary output to learn some more about **RMark**. The first part of the summary describes the type of model and some basic information. This simple analysis was for the CJS type of analysis and the model name defaults to the concatenation of the formulas used for each of the parameters in the model which was simply **Phi(~1)p(~1)**. The symbol ~ is used to begin a formula when the dependent variable on the left is not specified and implied. The "**1**" represents an intercept so "~ 1" is a model with only the intercept which is equivalent to the 'dot' in **MARK** notation. We will explain much more about specifying formulas later.

After the model description, **summary** provides the number of parameters in the model, the $-2\log(\mathcal{L})$ value and the AIC_c value for the model. We'll see later that the contents of this portion can vary depending on options for parameter counting and use of $\hat{c}$. Next the estimates, standard errors and confidence intervals for the β's are listed as they are shown similarly in the **MARK** output. The estimates are labeled with the type of parameter (e.g., **Phi** or **p** for CJS) and additional names related to the variables in the formula. For this model they are labeled intercept but later we'll see more informative labels. All of the labels for β and real parameters are done automatically and not manually by the user.

The real parameters are shown next in PIM format. For the CJS model, each of the parameters uses an upper-right triangular format for the PIM. The real values are shown for each parameter type (e.g., **Phi** and **p** in this case) and if there were groups defined, the values would be shown by group with a group label. The rows of the triangular PIMS are labeled with the cohort value (time of cohort release) and the columns are labeled with either the beginning time for time-interval parameters like **Phi** (survival

from time 1 to 2 is labeled with 1) or with the time for the occasion for occasion-specific parameters like **p** (re-capture probability at time i is labeled with i). This labeling is controlled by the value given to the beginning time of the experiment (**begin.time**) and by the lengths of the time intervals between occasions (**time.intervals**). For our simple example, we used the default of **begin.time=1** and all the time intervals being 1 so the rows are all labeled from 1 to 6 and the columns are labeled 1 to 6 for **Phi** to represent survival intervals $1 \rightarrow 2, 2 \rightarrow 3, \cdots, 6 \rightarrow 7$ and for **p** the columns are labeled 2 to 7 for the recapture occasions. Had we set **begin.time** to 1990, the rows and columns for **Phi** would have been labeled 1990 to 1995 and the columns for **p** would have been labeled 1991 to 1996.

By showing the real parameters in PIM format it can become readily obvious how the model is parameterized. Although this example is not a particularly good one, it is clear that the constant model was used as all of the real parameters are the same. We'll see more informative examples later.

A summary is nice and later we'll see other types of summaries but how do you look at the whole output file like you do in **MARK**? All you have to do is type the name of the object containing the results (e.g. **myexample**) and hit **enter**. **R** looks for a print method for the object when you type the name of the object (this is discussed in the **R** primer at the end of this appendix). When you type the name of an object with class **mark**, it will use the function **print.mark** to display the object. The function **print.mark** uses the Windows program **notepad.exe** to display the complete output file from **mark.exe**. Until you close the viewer you cannot continue in the **R** session that issued the call to the viewer. If you want to use a different program for viewing output files, simply assign the file specification for the program as a character string to the object **MarkViewer**.

Had we not assigned the results of **mark(dipper)** to **myexample**, **R** would have called **print.mark** to view the output file, and once it was closed, the summary output would have been displayed but no object would have been saved in the **R** workspace. However, the input and 3 output files would still be in the directory but they would not be linked to an object in the **R** workspace. If you were to make this mistake, you can create a **MARK** object in the **R** workspace and link the existing files to it by using the exact same call to **MARK** but adding the **filename** argument and specifying the base filename for the orphaned files. To see how this works, type **mark(dipper)** again but without assigning it to an object and it will create the files **mark002.***, because it is the second analysis that you have run. Then enter the following:

```
> myexample2=mark(dipper,filename="mark002")
```

he code will respond with a query when it sees that the files already exist.

```
Create MARK model with existing file (Y/N)?y
```

By entering "**y**" it will rebuild the model object and link the files to **myexample2**.

Occasionally you will run models and even create an **R** object for them but later decide to delete the **R** objects in the workspace. Deleting the **R** object will not delete the linked files. The function **cleanup** will purge orphaned input and output files. By typing **?cleanup** you will see the help file that describes this function. By typing **cleanup(ask=FALSE)**, all the orphaned files will be deleted. If you want to selectively delete the files, use **cleanup()** and you will be asked to confirm each file deletion.

To see how this works, remove **myexample2**, list the files, use **cleanup** and then list the files again as shown below:

```
> rm(myexample2)
> list.files()
 [1] "mark001.inp" "mark001.out" "mark001.res" "mark001.vcv" "mark002.inp"
```

```
 [6] "mark002.out" "mark002.res" "mark002.vcv"
> cleanup(ask=FALSE)
> list.files()
 [1] "mark001.inp" "mark001.out" "mark001.res" "mark001.vcv"
```

C.3. How RMark works

So now that you know how to create, summarize and print a simple model, let's learn more about how **RMark** works so you can fully understand the more realistic examples. To build and run the simple model for the dipper data you did not have to create PIMS nor a design matrix as you might in **MARK**, because **RMark** did it for you. But you may be saying to yourself, that is not really any different than the ability of the **MARK** interface to create pre-specified models. In some ways that is true but with a big difference. The **RMark** package widens the concept of pre-specified models to include user-defined formulas for model definition rather than the limited list of formulas in the **MARK** interface.

So how does it do that? Well with a few tricks and the **R** function **`model.matrix`**, it is surprisingly simple. The first trick is to realize that your options for developing models are limited by the PIM structure you choose and to fit completely general models without restrictions you need to use what **MARK** calls the *all-different* PIM structure. An all-different PIM is the default PIM type used in **RMark** (although there are some situations where it is useful to specify a simpler PIM structure - see section C.11). You can see the PIM structure by using the PIMS function for **`Phi`** and **`p`** with **`myexample`** as follows:

```
> PIMS(myexample,"Phi",simplified=FALSE)

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

> PIMS(myexample,"p",simplified=FALSE)

group = Group 1
   2  3  4  5  6  7
1 22 23 24 25 26 27
2    28 29 30 31 32
3       33 34 35 36
4          37 38 39
5             40 41
6                42
```

Each of the 21 real parameters in **`Phi`** (φ) and another 21 real parameters in **`p`** are given their own unique index, thus the term 'all-different'.

The second trick is realizing that you can *automatically* create and assign "*design data*" to the real parameters based on the model and group structure. This is truly the crux of **RMark** and what makes it possible to use formulae to create models. We use the term "design data" to represent "data" about

the model structure, or design. The design data that are created depends on the type of model (e.g, CJS, Multistrata) and the group structure. For a CJS model without groups, the "design data" are occasion (time), age and cohort-specific data. Separate design data are defined for each parameter (e.g., p and φ for CJS models) to allow flexibility and differences in the way design data are handled for each parameter. Also, for labeling it is better to keep them separate since some parameters like **Phi** represent an interval and others like **p** are for an occasion.

Using our first simple example let's describe the design data for the all-different PIMS shown above. There are many different kinds of design data that can be created for any particular example, but there are always several kinds of data that can be created automatically by default. For this example, they are **cohort**, **time** and **age**. We will first describe the design data for **p** which is represented by the indices 22 to 42. Imagine a table of data with 21 rows (one for each parameter) labeled 22 to 42. Let's define a cohort variable that represents the release cohort for each parameter. Rows 22-27 would contain a 1 because they are all for the first cohort, rows 28-32 would contain a 2,. . . , and row 42 would contain a 6. Likewise, if we wanted to create a time variable, then row 22 would contain a 2, rows 23 and 28 would contain a 3,. . . , and rows 27,32,36,39,41, and 42 would contain a 7 because all of those are in the last column for time 7. Likewise we can define a variable we'll call age which is really time-since-marking (TSM) unless all the animals are first released at the same age (e.g., banding young of the year birds). **Age(TSM)** is zero upon first release but it is 1 at the first recapture occasion and age is constant along the diagonals. To create an age variable, the rows 22,28,33,37,40 and 42 in our design data would each have a 1 in the age field, rows 23,29,34,38, and 41 would contain a 2,. . . , and row 27 would contain 6.

We will defer describing how the design data are actually created and can be manipulated but we will show you a summary and list of the first 10 rows of the design data for **p** beginning with index 22 of our design data object, that were created for **myexample** to explain the concept further.

```
group  cohort age   time      Cohort           Age             Time
 1:21   1:6    1:6   2:1   Min.   :0.000   Min.   :1.000   Min.   :0.000
        2:5    2:5   3:2   1st Qu.:0.000   1st Qu.:1.000   1st Qu.:2.000
        3:4    3:4   4:3   Median :1.000   Median :2.000   Median :4.000
        4:3    4:3   5:4   Mean   :1.667   Mean   :2.667   Mean   :3.333
        5:2    5:2   6:5   3rd Qu.:3.000   3rd Qu.:4.000   3rd Qu.:5.000
        6:1    6:1   7:6   Max.   :5.000   Max.   :6.000   Max.   :5.000

      group cohort age time Cohort Age Time
   22     1      1   1    2      0   1    0
   23     1      1   2    3      0   2    1
   24     1      1   3    4      0   3    2
   25     1      1   4    5      0   4    3
   26     1      1   5    6      0   5    4
   27     1      1   6    7      0   6    5
   28     1      2   1    3      1   1    1
   29     1      2   2    4      1   2    2
   30     1      2   3    5      1   3    3
   31     1      2   4    6      1   4    4
```

You will likely notice that there are more fields than we described and that some appear to be the same field. First off there is a group field that we didn't describe and it is always 1. This example did not have any group structure, thus all dippers were put in the same group numbered 1. We'll describe the use of grouping variables later. The **cohort**, **age** and **time** fields are created as factor variables as you can notice by the **summary** that shows the counts of the number of entries with each value (level) of

the variable. Then there are continuous versions of these variables named **Cohort**, **Age**, and **Time** which have been defined such that they start at 0 for **Cohort** and **Time**. In the **summary** the **min**, **max**, **mean** and **quartiles** are shown for these numeric variables. Capitalization was used to remain consistent with the **MARK** notation (actually, a Colorado State convention) of **p(t)** to represent fitting a model with a separate parameter for each occasion (level of time) and **p(T)** is a continuous trend with an intercept and slope as shown below. In **RMark**, these same models would be ~**time** and ~**Time** respectively.

So far this "trick" may just seem like added complication to the PIM concept. However, that is not the case once you know about the **R** function **model.matrix** which creates design matrices from a formula and data. Now that we have created "design data" for the real parameters, we only need to specify a formula using those data to create the design matrix. While you will never use **model.matrix** directly with the **RMark** package, it is useful to see a demonstration of it to understand how **RMark** works. It is also a useful way to check to make sure your model formula is correct. On the next page, we'll create the design matrix for the first 10 rows (representing parameters 22-31) for the following models for **p**: ~**time**, ~**Time**, ~**Time + age**:

```
> model.matrix(~time,myexample$design.data$p[1:10,])

   (Intercept) time3 time4 time5 time6 time7
1            1     0     0     0     0     0
2            1     1     0     0     0     0
3            1     0     1     0     0     0
4            1     0     0     1     0     0
5            1     0     0     0     1     0
6            1     0     0     0     0     1
7            1     1     0     0     0     0
8            1     0     1     0     0     0
9            1     0     0     1     0     0
10           1     0     0     0     1     0

 > model.matrix(~Time,myexample$design.data$p[1:10,])

   (Intercept) Time
1            1    0
2            1    1
3            1    2
4            1    3
5            1    4
6            1    5
7            1    1
8            1    2
9            1    3
10           1    4

 > model.matrix(~Time+age,myexample$design.data$p[1:10,])

   (Intercept) Time age2 age3 age4 age5 age6
1            1    0    0    0    0    0    0
2            1    1    1    0    0    0    0
3            1    2    0    1    0    0    0
4            1    3    0    0    1    0    0
```

```
5            1    4    0    0    0    1    0
6            1    5    0    0    0    0    1
7            1    1    0    0    0    0    0
8            1    2    1    0    0    0    0
9            1    3    0    1    0    0    0
10           1    4    0    0    1    0    0
```

Once the design data are defined, the **R** function does all the work of creating the design matrix for any formula using the design data. The design matrix is created using the convention called treatment contrasts. That means the first level is used as the intercept and the parameters for the remaining levels are an additive amount relative to the intercept. Also note that **`model.matrix`** automatically provides all of the label names for the β parameters as the column names of the design matrix.

If you only had these automatic design data, **RMark** would be fairly useful but would still be less than optimal. Later we'll show how you can manipulate and extend the design data to make it completely general and much more useful for designing models beyond these basic cookie-cutter types.

While all-different PIMS are necessary to enable creation of any model, they become problematic when the size of the problem is such that the number of real parameters (number of rows in the design matrix) exceeds 5,000. For some data sets and models this happens easily. Some large models will not run in **`mark.exe`** due to insufficient memory when the variance-covariance matrix for the real parameters is created. However, even if **MARK** can run the model it is quite inefficient and slow to use a design matrix with say 5000 real parameters and only 2 columns for the **`Phi(.)p(.)`** model.

This difficulty led to 'trick number 3' which is the concept of simplifying the design matrix. If you have the **`Phi(.)p(.)`** model with 5,000 real parameters, 2,500 of the design matrix rows would have a 1 in column 1 and a 0 in column 2 and the other 2,500 rows would have a 0 in column 1 and a 1 in column 2. That is quite redundant and really all one needs is the 2 unique rows to convey the information in the 5,000 rows. After the design matrix is created with the all-different PIMS, **RMark** simplifies it to contain only the unique rows and re-codes the PIMS. A link is maintained between the original indices and the new simplified indices. Simplification has important consequences for the viability of the modeling approach in **RMark** and the speed at which **`mark.exe`** completes the analysis.

To see the simplified and recoded PIMS for a model, you can use the **`PIMS`** function but this time using the default value of **`simplified=TRUE`**. If you use it with **`myexample`** as below you'll see that the 42 parameters have been recoded to the 2 unique parameters.

```
 > PIMS(myexample,"Phi")

group = Group 1
   1  2  3  4  5  6
1  1  1  1  1  1  1
2     1  1  1  1  1
3        1  1  1  1
4           1  1  1
5              1  1
6                 1

 > PIMS(myexample,"p")

group = Group 1
   2  3  4  5  6  7
1  2  2  2  2  2  2
```

```
2     2  2  2  2  2
3        2  2  2  2
4        2  2  2  2
5              2  2
6                 2
```

If you can simplify and recode the PIMS to the unique values, why would you want to keep the links to the original all-different indices? Because the original all-different PIMS provides a compatible foundation for all the analyses of the same data set using the same underlying type of model (e.g., CJS). With the all-different PIMS it is easier to display real parameters in PIM format, associate labels to the real parameters and to use model averaging on the real parameters from different models which will have different simplified PIM coding.

It should be helpful to examine the recoded PIMS for some other models, so without describing how we got them, we show the recoded PIMS for parameter **p** with ~**time**, ~**Time** and ~**Time+age** models with **Phi**(~**1**) as shown above for design matrices:

```
~time or ~Time group = Group 1
   2  3  4  5  6  7
1  2  3  4  5  6  7
2     3  4  5  6  7
3        4  5  6  7
4           5  6  7
5              6  7
6                 7


~Time + age group = Group 1
   2  3  4  5  6  7
1  2  3  4  5  6  7
2     8  9 10 11 12
3       13 14 15 16
4          17 18 19
5             20 21
6                22
```

Notice that the recoded PIMS for the ~**Time+age** model has 21 different parameters as with the all-different PIMS because with that model all of the rows of the design matrix for p are different. However, the PIM is recoded to start at 2 because **Phi(**~1**)** only requires a single parameter.

To a large extent the PIM/design simplification is transparent to you as a user in analyzing the data except that simplification does create a conflict between the labeling of real parameters in the **MARK** output and the labeling of real parameters in output from **summary** and other functions in **R**. When the PIMS are simplified there is no attempt to create a unique meaningful label for the real parameters in the input file sent to **mark.exe**. It uses the label associated with the first real parameter translated to the new PIM coding. However, the labeling of real parameters in **R** is maintained with the use of the all-different PIM structure. So use **R** when you want to look at real parameter values with their labels and ignore the labels in the **MARK** output file for real parameters.

PIM simplification is done for all parameters except for parameters that use the **mlogit** links like ψ in the multistrata model and **pent** in **POPAN**. The **mlogit** link assures that the sum of a specified

set of probabilities sums to 1 but it is implemented in **MARK** by using a sum of the unique real parameters indices and not the full set of real parameters. So for example, if you had 5 strata (**A** to **E**) and you wanted to estimate 4 real parameters for transitions from **A** by constraining equality for **D** and **E** ($\psi^{AB}, \psi^{AC}, \psi^{AD} = \psi^{AE}$). If you give these 4 parameters indices 1 to 4, then the **`mlogit`** link will work properly because it will sum across all 4, but if you give the parameters the indices 1,2,3,3 to constrain the last two parameters then the sum will be only the first 3 parameters and it will not sum the third parameter twice. Thus, an all-different PIM structure is required for parameters that use the **`mlogit`** link and any equality constraints must be implemented with the design matrix without any simplification of the PIMS. This restriction on **`mlogit`** links does not affect how you use **RMark** but may affect the speed at which **MARK** computes the parameter estimates because the number of parameters and the size of the design matrix is larger without PIM simplification.

As we showed above, **`model.matrix`** in **R** is the workhorse for creation of design matrices from formula; however, it cannot directly cope with individual covariates in the design matrix structure of **MARK** which uses the name of the individual covariate in the design matrix. To be generally useful, the formula notation needed to encompass individual covariates and this led to 'trick number 4' which is probably the only clever trick in the **RMark** implementation. But we'll delay divulging it until section C.16.

There are just a few things more you should understand before we move on. Note that the indices are "stacked on top of each other" to get unique indices for all of the parameters. Thus, for our example there are 21 φ parameters numbered 1 to 21 and 21 p parameters numbered 22 to 42. This ordering of the index numbers is done in a consistent fashion for each model. For example, p always follows φ in the CJS model. However, in most places in the code where you have to specify indices (see C.11 - fixing real parameters) it will typically only need to identify the parameter with the parameter-specific index which is the row number in the design matrix. Thus, in most cases for p, the parameters are identified by the indices 1 to 21. The only exception is situations in which you are referring to parameter indices across parameter types (e.g., both φ and p) as with the function **`covariate.predictions`** (C.16).

For most models in **MARK**, the design matrix could be displayed in the following manner:

design for parameter 1	0	0	0
0	*design for parameter 2*	0	0
0	0	$\ddots$	0
0	0	0	*design for parameter* ***k***

where none of the different types of parameters (e.g., p, φ etc) share columns of the design matrix. Parameter types can share the same covariate (e.g., φ_t p_t), but the effect of that covariate is not the same for the different types of parameters so the covariates are represented by different columns in the design matrix. For most models, this works quite well but there are some exceptions including parameters "`p`" and "`c`" in the closed and robust design models, parameters "`p1`" and "`p2`" in the **`MSOccupancy`** model, and "`GammaPrime`" and "`GammaDoublePrime`" in the robust design models. In each of these cases the parameter has a different name but it is effectively the same type of parameter, so it is quite reasonable to build models in which they "share" covariates or are equated. To accommodate this exception, the parameter listed first is set as the dominant parameter and the formula for the dominant parameter is given a special argument "**`share`**" that can be set to **`TRUE`** or **`FALSE`**. If it is set to **`TRUE`**, then the design

data are combined 'on the fly' and an extra column is added for the non-dominant parameter to enable fitting additive models. See section C.19 for an example.

C.4. Dissecting the function "mark"

Now that you have been introduced to some of the ideas on the inner workings of **RMark** like design data and PIM structure and simplification, we'll discuss the steps that are taken in producing an analysis and along the way we will expand the concept of design data to include group structure. The function **mark** is actually quite simple because it is a convenience function that calls 5 other functions that actually do the work in the following order:

1. **process.data**
2. **make.design.data**
3. **make.mark.model**
4. **run.mark.model**
5. **summary.mark**

Why do you care? Primarily because the function has dual calling modes for efficiency and to enable adding/modifying the design data. Depending on the arguments that you pass **mark**, it will either start with **process.data** or it will skip directly to **make.mark.model**. This allows you to do the first 2 steps once, optionally modify the design data, and then run a whole series of models on the data without repeating the first 2 steps in each call to **mark**.

C.4.1. Function process.data

The first function **process.data** literally does what its name implies. It takes the input data frame and the user-defined arguments and creates a list (processed data) containing the data and numerous defined attributes that the remaining functions use in defining the analysis models. The following are the primary attributes that are set:

1. **model**: the type of analysis model (e.g., "CJS", "Known", "POPAN"); see help for function **mark (?mark)** for a complete listing of the supported models
2. **begin.time**: the time of the first capture/release occasion for labeling
3. **time.intervals**: the lengths of the time intervals between capture occasions
4. **groups**: the list of factor variables in the data to define groups
5. **initial.ages**: the age of animals at first capture/release corresponding to the levels of the age grouping variable (**age.var**)
6. **nocc**: number of capture/encounter occasions which is determined from the contents of the "**ch**" field in the data and the type of analysis **model(model)**.

As an example, we will use the dipper data and the field **sex** to create 2 groups in the data and define fictitious beginning time and time intervals for the data:

```
> data(dipper)
> dipper.process=process.data(dipper,model="CJS",begin.time=1980,
                              time.intervals=c(1,.5,1,.75,.25,1),groups="sex")
```

The resulting object (**dipper.process**) is a list containing the data and its attributes. The names of the elements of the list can be viewed with the **names** function:

```
> names(dipper.process)
 [1] "data"             "model"            "mixtures"         "freq"
 [5] "nocc"             "nocc.secondary"   "time.intervals"   "begin.time"
 [9] "age.unit"         "initial.ages"     "group.covariates" "nstrata"
[13] "strata.labels"
```

Note that there are many more attributes than described above. Some – like **mixtures**, **nstrata**, **nocc.secondary** and **strata.labels** – are only relevant to specific models but these are often included with a default, NULL or empty value for models in which they are not relevant. Specific elements of the list can be extracted as illustrated:

```
> dipper.process$nocc
[1] 7
> dipper.process$group.covariates
     sex
1 Female 2   Male
> dipper.process$begin.time
[1] 1980
> dipper.process$strata.labels
character(0)
> dipper.process$nocc.secondary
NULL
> dipper.process$time.intervals
[1] 1.00 0.50 1.00 0.75 0.25 1.00
```

From the first 5 rows of the field **freq** it is obvious that this is the structure used to create the frequency data for the **MARK** input file with the defined grouping structure and the column labels as the group labels:

```
> dipper.process$freq[1:10,]
   sexFemale sexMale
1          1       0
2          1       0
3          1       0
4          1       0
5          1       0
```

The structure of the encounter history and the analysis depends on the analysis model that you choose like "**CJS**" above. Thus, it is necessary to process the data frame (data) containing the encounter history and a chosen model to define the relevant values which will be used by the remaining functions. For example, number of capture occasions (**nocc**) is automatically computed based on the length of the encounter history (**ch**) in data; however, this is dependent on the type of analysis model. For models such as "**CJS**", "**Pradel**" and others, it is simply the length of **ch**. Whereas, for "**Burnham**" and "**Barker**" models, the encounter history contains capture and resight/recovery values so **nocc** is one-half the length of **ch**. Likewise, the number of **time.intervals** depends on the model. For models, such as "**CJS**", "**Pradel**" and others, the number of **time.intervals** is **nocc**-1; whereas, for capture-recovery (or resight) models the number of **time.intervals** is **nocc**. The default time interval is unit time (1) and if

this is adequate, the function will assign the appropriate length; otherwise the appropriate number of values must be given.

A processed data frame can only be analyzed using the **model** that was specified in the call to **process.data**. The **model** value is used by the functions **make.design.data** and **make.mark.model** to define the design data and the appropriate input file structure for **MARK**. Thus, if the data are going to be analyzed with different underlying models, create different processed data objects possibly using the type of **model** as an extension. For example,

```
dipper.cjs=process.data(dipper,model="CJS")
dipper.popan=process.data(dipper,model="POPAN")
```

The **process.data** function will report any inconsistencies in the lengths of the capture history values and when invalid entries are given in the capture history. For example, with the "**CJS**" model, the capture history should only contain 0 and 1 whereas for "**Barker**" it can contain 0,1,2. For "**Multistrata**" models, the code will automatically identify the number of strata (**nstrata**) and strata labels (**strata.labels**) based on the unique alphabetic codes used in the capture histories. For "**Robust**" design models, the number of secondary occasions (**nocc.secondary**) is determined by the specified **time.intervals**.

The argument **begin.time** specifies the time for the first capture/release occasion. This is used in creating the levels of the **time** factor variable in the design data and for labeling parameters. If **begin.time** varies by group, enter a vector of times with one for each group.

The argument groups can contain one or more character strings specifying the names of factor variables contained in **data**. A group is created for each unique combination of the levels of the factor variables. Further examples of grouping and use of age variables will be given later and they can be found in the help documentation with **R** (**?process.data** and **?example.data**).

C.4.2. Function `make.design.data`

The next step is to create the design data and PIM structure which depends on the selected type of analysis model (e.g., CJS or Multistrata), number of occasions, grouping variables and other attributes of the data that were defined in the processed data, which is the first and primary argument to the function **make.design.data** that creates the design data. For parameters with triangular PIMS the default design data are **cohort**, **age** and **time** and any grouping factor variables that were defined. For parameters with square PIMS, there is only one row so the cohort variable is not automatically included in the design data but there are ways to create a cohort structure in this case with groups.

In creating the factor variables for cohort, age, and time, a separate factor level is created for each value of the variable. However, you can optionally bin the values into intervals in creating the factor variable. For example, if birds were always classified as either young (< 1) or as adult (1+), then **age.bins** could be specified in the call to **make.design.data**. However, if you wanted the option to model age based on all levels of the factor and other models with some ages collapsed into intervals then it is best to allow **make.design.data** to create the default factor variables and create additional design data with the function **add.design.data** or using **R** statements and functions. There are many other features of **make.design.data** including restricting parameters to use "time" or "constant" PIMS, setting the subtraction stratum for "Multistrata" models, and automatic removal of unused design data. These features are described in the help files (**?make.design.data** and **?add.design.data**) and they are described in more detail in later sections.

For now, a simple example with the dipper data will suffice to illustrate this step and explain the basic concepts. But before we do that we'll reprocess the data to use annual time intervals rather than the fictitious ones used above:

```
> dipper.process=process.data(dipper,model="CJS",begin.time=1980,groups="sex")
```

The result of a call to **`make.design.data`** is a list of design data, so one naming convention is to use **`ddl`** (design data list) as the suffix and the data name as the prefix as follows:

```
> dipper.ddl=make.design.data(dipper.process)
```

Before we look at the design data, let's run a simple model with **`mark`** but this time rather than specifying the data file, we'll specify the processed data and the design data list. When **MARK** is called with these 2 arguments it recognizes that they have already been created and skips to step 3 to create and run the model directly.

```
> myexample2=mark(dipper.process,dipper.ddl)

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

Npar :  2
-2lnL:  666.8377
AICc :  670.866

Beta
                  estimate        se       lcl       ucl
Phi:(Intercept) 0.2421484 0.1020127 0.0422035 0.4420933
p:(Intercept)   2.2262658 0.3251093 1.5890517 2.8634800


Real Parameter Phi Group:sexFemale
      1980     1981     1982     1983     1984     1985
1980 0.560243 0.560243 0.560243 0.560243 0.560243 0.560243
1981          0.560243 0.560243 0.560243 0.560243 0.560243
1982                   0.560243 0.560243 0.560243 0.560243
1983                            0.560243 0.560243 0.560243
1984                                     0.560243 0.560243
1985                                              0.560243

Group:sexMale
         1980     1981     1982     1983     1984     1985
1980 0.560243 0.560243 0.560243 0.560243 0.560243 0.560243
1981          0.560243 0.560243 0.560243 0.560243 0.560243
1982                   0.560243 0.560243 0.560243 0.560243
1983                            0.560243 0.560243 0.560243
1984                                     0.560243 0.560243
1985                                              0.560243


Real Parameter p Group:sexFemale
          1981      1982      1983      1984      1985      1986
1980 0.9025835 0.9025835 0.9025835 0.9025835 0.9025835 0.9025835
1981           0.9025835 0.9025835 0.9025835 0.9025835 0.9025835
1982                     0.9025835 0.9025835 0.9025835 0.9025835
```

```
1983                                0.9025835 0.9025835 0.9025835
1984                                          0.9025835 0.9025835
1985                                                    0.9025835

Group:sexMale
          1981      1982      1983      1984      1985      1986
1980 0.9025835 0.9025835 0.9025835 0.9025835 0.9025835 0.9025835
1981           0.9025835 0.9025835 0.9025835 0.9025835 0.9025835
1982                     0.9025835 0.9025835 0.9025835 0.9025835
1983                               0.9025835 0.9025835 0.9025835
1984                                         0.9025835 0.9025835
1985                                                   0.9025835
```

If you are following along with these commands and did not get the results above make sure that you reprocessed the data with the annual intervals and then created the design data before entering the call to **mark** because the results will vary with different time intervals. Notice that the results are exactly the same as the first analysis we did with the dipper data; however, the real parameter summaries are displayed for each sex because it was used to define groups.

Now let's look at the non-simplified PIMS for φ and compare them to the design data that were created.

```
> PIMS(myexample2,"Phi",simplified=FALSE)

group = sexFemale
     1980 1981 1982 1983 1984 1985
1980    1    2    3    4    5    6
1981         7    8    9   10   11
1982              12   13   14   15
1983                   16   17   18
1984                        19   20
1985                             21

group = sexMale
     1980 1981 1982 1983 1984 1985
1980   22   23   24   25   26   27
1981        28   29   30   31   32
1982             33   34   35   36
1983                  37   38   39
1984                       40   41
1985                            42
```

To accommodate the group structure 42 possible real parameter indices were created for **Phi** with 1-21 for females and 22-42 for males. The same structure was also created for **p**. If we look at the names of the design data list

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

we see that there are 3 elements in the list. The first 2 are the design data for the parameters in the CJS model (**Phi** and **p**) and the last is a list of the type of PIMS used which in this case is the default of all-different. We can examine the design data for **Phi** as follows (with abbreviated output):

```
> dipper.ddl$Phi
   group cohort age time Cohort Age Time    sex
1  Female   1980   0 1980      0   0    0 Female
2  Female   1980   1 1981      0   1    1 Female
3  Female   1980   2 1982      0   2    2 Female
4  Female   1980   3 1983      0   3    3 Female
5  Female   1980   4 1984      0   4    4 Female
6  Female   1980   5 1985      0   5    5 Female
7  Female   1981   0 1981      1   0    1 Female
8  Female   1981   1 1982      1   1    2 Female
9  Female   1981   2 1983      1   2    3 Female
10 Female   1981   3 1984      1   3    4 Female
<...>
22   Male   1980   0 1980      0   0    0   Male
23   Male   1980   1 1981      0   1    1   Male
37   Male   1983   0 1983      3   0    3   Male
38   Male   1983   1 1984      3   1    4   Male
39   Male   1983   2 1985      3   2    5   Male
40   Male  1984    0 1984      4   0    4   Male
41   Male   1984   1 1985      4   1    5   Male
42   Male   1985   0 1985      5   0    5   Male
```

Rows (indices) 1 and 22 have the same design data except that row 1 is for females and row 22 is for males. Any grouping variables are automatically included into the design data. A field "group" is created which is a unique combination of the different values of each grouping variable and then a separate field is included for each grouping variable. With a single grouping variable, like sex, the 2 fields are identical. The pre-defined models in **MARK** like $\{\varphi_{g*t}\}$ are equivalent to using the field group in the formula. As we will show later, the inclusion of each grouping variable allows additive models to be created with the grouping variables rather than just using the group field which is the full interaction of the grouping variables.

C.5. More simple examples

Hopefully you now have a basic understanding of how PIMS, design data and design matrices are created in **RMark** and we can move on to learning how to specify formula for analysis models. Along the way we'll reiterate and expand on the material we have presented so far. We will continue on with the dipper data with **`sex`** used for groups to describe using **`formula`** with the existing design data created by default and then we'll consider examples that work with user-defined supplemental design data.

Following along the lines of **MARK**, a model is described by sub-models for each parameter of the particular type of mark-recapture analysis. With the dipper data we have been using the CJS model with parameters φ and p, and so far we have been using the default model which is a constant value for each parameter. A parameter specification (sub-model) is defined by a list, although in most circumstances the list will only contain a single element named the **`formula`**. For reasons that will be obvious later, the parameter specifications should be assigned to an object named with a prefix being the parameter name and the suffix being a description for the formula or some other strategy like numbering. For example, with the simple model we have constructed so far the parameter specifications would be:

```
> Phi.dot=list(formula=~1)
> p.dot=list(formula=~1)
```

The parameter specifications are used with the **mark** argument **model.parameters** to define the model. The default model we ran earlier could also be specified as:

```
> myexample2=mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.dot,p=p.dot))
```

Now the parameter specification **Phi.dot** and **p.dot** are identical so you could have done the following:

```
> myexample2=mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.dot,p=Phi.dot))
```

and gotten the same results but that could be a bit confusing and later you'll see that there are advantages to having a separate parameter specification object for each parameter even if they have the same values.

So, let's create some more parameter specifications solely for demonstration purposes as some of these models may not make sense for the dipper data:

```
Phi.time=list(formula=~time)
Phi.sex=list(formula=~sex)
Phi.sexplusage=list(formula=~sex+age)
p.time=list(formula=~time)
p.Time=list(formula=~Time)
p.Timeplussex=list(formula=~Time+sex)
```

By including the dot models, we could easily specify 16 (4×4) different models for all the combinations of these parameter specifications. Hmm, how do we name all these models to keep them straight? One way is to use the data name and add on the parameter specifications as in the following examples:

```
dipper.phi.dot.p.dot=
          mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.dot,p=p.dot))
dipper.phi.time.p.dot=
          mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.time,p=p.dot))
dipper.phi.sex.p.dot=
          mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.sex,p=p.dot))
dipper.phi.sex.p.Timeplussex=
      mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.sex,p=p.Timeplussex))
dipper.phi.time.p.time=
          mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.time,p=p.time))
dipper.phi.sexplusage.p.dot=
      mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.sexplusage,p=p.dot))
```

See how easy it is? No messing with PIMS or design matrices. You are certainly getting the idea but let's look at the last 3 models in more detail to learn some more. If you ran these by simply copying the text into **R**, the output will have passed by on the screen but we can simply repeat it with the **summary** function:

```
 > summary(dipper.phi.sex.p.Timeplussex)

Output summary for CJS model
Name : Phi(~sex)p(~Time + sex)

Npar :  5
-2lnL:  664.1672
AICc :  674.3101
```

```
Beta
                 estimate        se        lcl       ucl
Phi:(Intercept) 0.1947163 0.1403108 -0.0802928 0.4697254
Phi:sexMale     0.7547928 0.1989333 -0.3351165 0.4447022
p:(Intercept)   1.2297543 0.6455548 -0.0355331 2.4950417
p:Time          0.3162690 0.2255297 -0.1257693 0.7583073
p:sexMale       0.4290287 0.6660079 -0.8763468 1.7344042

Real Parameter Phi Group:sexFemale
          1980      1981      1982      1983      1984      1985
1980 0.5485258 0.5485258 0.5485258 0.5485258 0.5485258 0.5485258
1981           0.5485258 0.5485258 0.5485258 0.5485258 0.5485258
1982                     0.5485258 0.5485258 0.5485258 0.5485258
1983                               0.5485258 0.5485258 0.5485258
1984                                         0.5485258 0.5485258
1985                                                   0.5485258

Group:sexMale
          1980      1981      1982      1983      1984      1985
1980 0.5620557 0.5620557 0.5620557 0.5620557 0.5620557 0.5620557
1981           0.5620557 0.5620557 0.5620557 0.5620557 0.5620557
1982                     0.5620557 0.5620557 0.5620557 0.5620557
1983                               0.5620557 0.5620557 0.5620557
1984                                         0.5620557 0.5620557
1985                                                   0.5620557


Real Parameter p Group:sexFemale
          1981      1982      1983      1984      1985      1986
1980 0.7737756 0.8243386 0.8655639 0.8983077 0.9237786 0.9432727
1981           0.8243386 0.8655639 0.8983077 0.9237786 0.9432727
1982                     0.8655639 0.8983077 0.9237786 0.9432727
1983                               0.8983077 0.9237786 0.9432727
1984                                         0.9237786 0.9432727
1985                                                   0.9432727

Group:sexMale
          1981      1982      1983      1984      1985      1986
1980 0.8400746 0.8781527 0.9081557 0.9313485 0.9490134 0.9623168
1981           0.8781527 0.9081557 0.9313485 0.9490134 0.9623168
1982                     0.9081557 0.9313485 0.9490134 0.9623168
1983                               0.9313485 0.9490134 0.9623168
1984                                         0.9490134 0.9623168
1985                                                   0.9623168
```

Let's look at the **mark** result object some more so you can see how to extract various parts of the results. We see that the names of the elements are:

```
> names(dipper.phi.sex.p.Timeplussex)
```

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

The field **output** is the link to the input and output files

```
> dipper.phi.sex.p.Timeplussex$output
 [1] "mark012"
```

The value may be different for you depending on how many models you have run and whether you removed models and used the **cleanup** function. The element **pims** is the all-different PIMS for the model but the extractor function **PIMS** produces clearer output than simply typing the command **dipper.phi.sex.p.Timeplussex$pims**. The element **model.parameters** is simply the value of the **mark** argument with the same name; whereas, the **parameters** field is for internal use with various attributes set for each parameter. Likewise, the links between the simplified PIMS and the non-simplified PIMS contained in the list element **simplify** is only useful internally. The design matrix for the simplified model structure is also contained in the result as a matrix:

```
> dipper.phi.sex.p.Timeplussex$design.matrix
                          Phi:(Intercept) Phi:sexMale p:(Intercept) p:Time p:sexMale
Phi gFemale c1980 a0 t1980 "1"            "0"         "0"           "0"    "0"
Phi gMale c1980 a0 t1980   "1"            "1"         "0"           "0"    "0"
p gFemale c1980 a1 t1981   "0"            "0"         "1"           "0"    "0"
p gFemale c1980 a2 t1982   "0"            "0"         "1"           "1"    "0"
p gFemale c1980 a3 t1983   "0"            "0"         "1"           "2"    "0"
p gFemale c1980 a4 t1984   "0"            "0"         "1"           "3"    "0"
p gFemale c1980 a5 t1985   "0"            "0"         "1"           "4"    "0"
p gFemale c1980 a6 t1986   "0"            "0"         "1"           "5"    "0"
p gMale c1980 a1 t1981     "0"            "0"         "1"           "0"    "1"
p gMale c1980 a2 t1982     "0"            "0"         "1"           "1"    "1"
p gMale c1980 a3 t1983     "0"            "0"         "1"           "2"    "1"
p gMale c1980 a4 t1984     "0"            "0"         "1"           "3"    "1"
p gMale c1980 a5 t1985     "0"            "0"         "1"           "4"    "1"
p gMale c1980 a6 t1986     "0"            "0"         "1"           "5"    "1"
```

The list element of most interest is **results**, a list containing extracted values from the **MARK** output files:

```
> names(dipper.phi.sex.p.Timeplussex$results)
[1] "lnl"              "deviance"         "npar"             "n"
[5] "AICc"             "beta"             "real"             "beta.vcv"
[9] "derived"          "derived.vcv"      "covariate.values" "singular"
```

The definitions of the elements are as follows:

- **lnl**: $-2\log\mathcal{L}$ Likelihood value
- **deviance**: difference between null deviance and model deviance
- **npar**: Number of parameters (always the number of columns in design matrix)

- n: effective sample size
- **AICc**: Small sample corrected AIC using **npar**
- **beta**: data frame of β parameters with estimate, standard error (**se**), lower confidence limit (**lcl**), and upper confidence limit (**ucl**)
- **real**: data frame of unique (simplified) real parameters with estimate, standard error (**se**), lower confidence limit (**lcl**), and upper confidence limit (**ucl**), and notation for fixed parameters
- **beta.vcv**: variance-covariance matrix for β
- **derived**: dataframe of derived parameters if any
- **derived.vcv**: variance-covariance matrix for derived parameters if any
- **covariate.values**: dataframe with fields **Variable** and **Value** which are the covariate names and value used for real parameter estimates in the **MARK** output
- **singular**: indices of β parameters that are non-estimable or at a boundary

The individual elements can be extracted using list notation. For example, the data frame of the β parameters:

```
> dipper.phi.sex.p.Timeplussex$results$beta
                 estimate        se        lcl       ucl
Phi:(Intercept) 0.1947163 0.1403108 -0.0802928 0.4697254
Phi:sexMale     0.7547928 0.1989333 -0.3351165 0.4447022
p:(Intercept)   1.2297543 0.6455548 -0.0355331 2.4950417
p:Time          0.3162690 0.2255297 -0.1257693 0.7583073
p:sexMale       0.4290287 0.6660079 -0.8763468 1.7344042
```

or the data frame of the unique (simplified) real parameters:

```
> dipper.phi.sex.p.Timeplussex$results$real
                            estimate        se       lcl       ucl fixed
Phi gFemale c1980 a0 t1980 0.5485258 0.0347473 0.4799376 0.6153188
Phi gMale c1980 a0 t1980   0.5620557 0.0349965 0.4927116 0.6290571
p gFemale c1980 a1 t1981   0.7737756 0.1130024 0.4911177 0.9237935
p gFemale c1980 a2 t1982   0.8243386 0.0721225 0.6387191 0.9256861
p gFemale c1980 a3 t1983   0.8655639 0.0495235 0.7365526 0.9368173
p gFemale c1980 a4 t1984   0.8983077 0.0424479 0.7803679 0.9564497
p gFemale c1980 a5 t1985   0.9237786 0.0418005 0.7910491 0.9748739
p gFemale c1980 a6 t1986   0.9432727 0.0411246 0.7866317 0.9868417
p gMale c1980 a1 t1981     0.8400746 0.0970652 0.5603827 0.9558434
p gMale c1980 a2 t1982     0.8781527 0.0627026 0.6956116 0.9578565
p gMale c1980 a3 t1983     0.9081557 0.0430622 0.7823503 0.9645393
p gMale c1980 a4 t1984     0.9313485 0.0345158 0.8248454 0.9750509
p gMale c1980 a5 t1985     0.9490134 0.0312841 0.8397867 0.9850955
p gMale c1980 a6 t1986     0.9623168 0.0291539 0.8408254 0.9919649
```

Remember that the labels for the real parameters in the simplified model can be misleading due to the simplification process.

To view all of the real parameters with standard errors, use **summary** as follows (the output has been abbreviated):

```
> summary(dipper.phi.sex.p.Timeplussex,se=T)

Output summary for CJS model Name : Phi(~sex)p(~Time + sex)

Real Parameter p
                          par.index  estimate        se       lcl       ucl fixed
p gFemale c1980 a1 t1981          3 0.7737756 0.1130024 0.4911177 0.9237935
p gFemale c1980 a2 t1982          4 0.8243386 0.0721225 0.6387191 0.9256861
p gFemale c1980 a3 t1983          5 0.8655639 0.0495235 0.7365526 0.9368173
p gFemale c1980 a4 t1984          6 0.8983077 0.0424479 0.7803679 0.9564497
p gFemale c1980 a5 t1985          7 0.9237786 0.0418005 0.7910491 0.9748739
p gFemale c1980 a6 t1986          8 0.9432727 0.0411246 0.7866317 0.9868417
p gFemale c1981 a1 t1982          4 0.8243386 0.0721225 0.6387191 0.9256861
p gFemale c1981 a2 t1983          5 0.8655639 0.0495235 0.7365526 0.9368173
p gFemale c1981 a3 t1984          6 0.8983077 0.0424479 0.7803679 0.9564497
p gFemale c1981 a4 t1985          7 0.9237786 0.0418005 0.7910491 0.9748739
p gFemale c1981 a5 t1986          8 0.9432727 0.0411246 0.7866317 0.9868417
p gFemale c1982 a1 t1983          5 0.8655639 0.0495235 0.7365526 0.9368173
p gFemale c1982 a2 t1984          6 0.8983077 0.0424479 0.7803679 0.9564497
p gFemale c1982 a3 t1985          7 0.9237786 0.0418005 0.7910491 0.9748739
p gFemale c1982 a4 t1986          8 0.9432727 0.0411246 0.7866317 0.9868417
p gFemale c1983 a1 t1984          6 0.8983077 0.0424479 0.7803679 0.9564497
p gFemale c1983 a2 t1985          7 0.9237786 0.0418005 0.7910491 0.9748739
p gFemale c1983 a3 t1986          8 0.9432727 0.0411246 0.7866317 0.9868417
p gFemale c1984 a1 t1985          7 0.9237786 0.0418005 0.7910491 0.9748739
p gFemale c1984 a2 t1986          8 0.9432727 0.0411246 0.7866317 0.9868417
p gFemale c1985 a1 t1986          8 0.9432727 0.0411246 0.7866317 0.9868417
p gMale   c1980 a1 t1981          9 0.8400746 0.0970652 0.5603827 0.9558434
p gMale   c1980 a2 t1982         10 0.8781527 0.0627026 0.6956116 0.9578565
p gMale   c1980 a3 t1983         11 0.9081557 0.0430622 0.7823503 0.9645393
p gMale   c1980 a4 t1984         12 0.9313485 0.0345158 0.8248454 0.9750509
```

The **par.index** field is the index within the simplified set of real parameters (i.e., the recoded parameter index). The label for the real parameter uses a short hand notation in which **g** is for group, **c** for cohort, **a** for age and **t** for time. After each letter is the value of the variable. In other types of mark-recapture models, like **Multistrata**, additional values are added like s for stratum, and **t** for **to stratum**, for movement from one stratum to another stratum.

C.6. Design covariates in RMark

There are 2 types of covariates used in **RMark**. You have already seen examples of the first type which is the design covariate (design data). Design covariates are linked to the parameters in the model and specify differences in the parameters associated with the model structure (e.g., time, cohort) or with group structure of the animals (e.g., sex) because different parameters are used for different groups of animals. The second type of covariate is individual covariates which specify differences in the individual animals. The distinction between the types is not entirely clear-cut because design covariates for group

structure are individual covariates because each animal has its own value. However, group design covariates have 2 important restrictions: 1) they must be a factor variable which means they will typically have a small number of unique values (e.g., sex=M or F), and 2) the value cannot change over time. Thus, individual covariates are typically used for numeric variables (e.g., mass, length) or for covariates where the value changes over time (e.g., trap dependence). You can code factor variables as individual covariates by creating $(k-1)$ dummy variables (0/1) for a factor variable with k levels, but it is usually better to use factor variables as group design covariates. Design covariates are stored in the design data (`ddl`) and individual covariates remain with the encounter history data. Use of individual covariates in data and models is described in C.16 and in this section we demonstrate how the flexibility of design covariates can be used to expand the usefulness of model formula.

So far, all of the examples we have created have only used the design data created by default using the group and model structure. While that may be all that is needed in many instances, additional design data can be created and used in formula and this substantially adds to the flexibility of model development. What kinds of design data can be added and why would you want to do that? Any data that are relevant to the model and group structure can be added to the design data. These can be dummy variables that enable "effects" to be modeled for subsets of any of the design data fields. For example, below we will create a design data field called **`Flood`** for the dipper example which is 1 in years with floods and 0 in non-flood years. Dummy variables are equivalent to coding a column in the design matrix as you do with the standard **MARK** interface. Or the added design data fields may create a factor variable with new intervals of existing design data. For example, we'll create a design data field that bins ages as young (0 and 1) and sub-adult (2-3), and adult (4^+) (*note*: this and other treatments of the dipper data may not be realistic for dippers). Finally, the added design data could be a numeric field that is specific to some parameter. For example, we'll create an effort field for each sampling occasion in the dipper data to model capture probability.

Design data can be created and modified with any relevant **R** statement or function. We will start with a simple example using the dipper data using the fictitious dates we assigned. With the dipper data, between sampling occasions 1981-1982 and 1982-1983 there were severe floods that could have reduced survival in those periods and capture probability may have differed in 1982 (note: use of these dates may not reflect the true situation). To model this effect, we will define a **`Flood`** variable that is 1 for flood periods and 0 otherwise. Remember that there are different design data for each parameter, so a **`Flood`** field has to be defined for each parameter that will use the field in the model. Because the timing of the effect varies for φ and p, the definitions of those variables are different.

```
> dipper.ddl$Phi$Flood=0
> dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==1981 |  dipper.ddl$Phi$time==1982]=1
> dipper.ddl$p$Flood=0
> dipper.ddl$p$Flood[dipper.ddl$p$time==1982]=1
```

The first statement above creates a **`Flood`** field for **`Phi`** and assigns the value 0 to all values. The second statement assigns 1 to **`Flood`** for those rows in the dataframe for which time is either 1981 (for interval 1981 to 1982) or 1982 (for interval 1982 to 1983). The last 2 statements define the **`Flood`** variable for capture probability. Once the data have been created they can be used in models as shown below:

```
> Phi.Flood=list(formula=~Flood)
> p.Flood=list(formula=~Flood)
> dipper.phi.flood.p.dot=
     mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.Flood,p=p.dot))
> dipper.phi.flood.p.flood=
     mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.Flood,p=p.Flood))
```

While you can use any **R** statement to create design data, in many instances the design data you are creating is a modification of existing data or merges new data with existing data, so some functions were created to simplify the process. If the new design data are simply creating bins (intervals) of time, age, or cohort, then you can use the function **add.design.data**. For example, if we want to create age intervals for survival (young, sub-adult, and adult) as we described above, we can do it as follows:

```
> dipper.ddl=add.design.data(dipper.process, dipper.ddl,
    parameter="Phi", type="age", bins=c(0,1,3,6),name="ageclass")
```

If we summarize the design data for **Phi**, we see that the variable we chose to name **ageclass** has been defined properly:

```
> summary(dipper.ddl$Phi)
    group      cohort    age       time         Cohort            Age
 Female:21   1980:12   0:12   1980: 2   Min.   :0.000   Min.   :0.000
 Male  :21   1981:10   1:10   1981: 4   1st Qu.:0.000   1st Qu.:0.000
             1982: 8   2: 8   1982: 6   Median :1.000   Median :1.000
             1983: 6   3: 6   1983: 8   Mean   :1.667   Mean   :1.667
             1984: 4   4: 4   1984:10   3rd Qu.:3.000   3rd Qu.:3.000
             1985: 2   5: 2   1985:12   Max.   :5.000   Max.   :5.000
      Time           sex          Flood          ageclass
 Min.   :0.000   Female:21   Min.   :0.0000   [0,1]:22
 1st Qu.:2.000   Male  :21   1st Qu.:0.0000   (1,3]:14
 Median :4.000               Median :0.0000   (3,6]: 6
 Mean   :3.333               Mean   :0.2381
 3rd Qu.:5.000               3rd Qu.:0.0000
 Max.   :5.000               Max.   :1.0000
```

It is always a good idea to examine the design data after you have created it to make sure that the intervals were defined as expected and that they included the entire range of the data. In the definition of **ageclass**, a "(" means the interval is open on the left which means that value is not included in the interval. Whereas a square bracket ("[" or "]") is for a closed interval which means the interval end point is included. If we decided that the intervals should be shifted to the left, the easiest way is as follows:

```
> dipper.ddl=add.design.data(dipper.process, dipper.ddl,
    parameter="Phi", type="age",
       bins=c(0,1,3,6),name="ageclass",right=FALSE,replace=TRUE)
```

Had we not used **replace=T**, we would have gotten the following error:

```
Error in add.design.data(dipper.process, dipper.ddl, parameter =
"Phi",  : Variable  ageclass  already in design data. Use
replace=TRUE if you want to replace current values
```

Now **ageclass** defines the intervals 0,1 to 2, and 3+ for modeling age effects in **Phi**:

```
> summary(dipper.ddl$Phi$ageclass)
[0,1) [1,3) [3,6]
   12    18    12
```

Had we issued the following function call:

```
> dipper.ddl=add.design.data(dipper.process, dipper.ddl,
    parameter="Phi", type="age",bins=c(1,3,6),name="badageclass")
```

then when we summarized the field, the presence of **NA**s make it apparent that the defined bins did not span the range of the **age** field:

```
> summary(dipper.ddl$Phi$badageclass)
[1,3] (3,6]  NA's
   24     6    12
```

The NAs occurred in this case because 0 was excluded. Notice that the intervals are always closed on the far left and far right. Since we do not want this field, by assigning **NULL** to the field

```
> dipper.ddl$Phi$badageclass=NULL
```

it is removed from the design data for Phi:

```
> names(dipper.ddl$Phi)
 [1] "group"    "cohort"   "age"   "time"  "Cohort"   "Age"   "Time"   "sex"
 [8] "Flood"    "ageclass"
```

In many situations the additional design data are simply covariates to be used in place of occasion/-time effects. Examples are effort, weather, or observers which vary for occasions and may be useful to simplify modeling of capture probability rather than time-varying parameters. For this situation, the function **merge_design.covariates** was created. The following is an example in which fictitious effort data were created for the dipper data:

```
> df=data.frame(time=c(1980:1986),effort=c(10,5,2,8,1,2,3))
> dipper.ddl$p=merge_design.covariates(dipper.ddl$p,Xdf)
> summary(dipper.ddl$p$effort)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   2.000   2.000   3.095   3.000   8.000
```

So why is the maximum value for effort only 8 and not 10? For the CJS model there is no capture probability for 1980, so the value is ignored. The function is less forgiving if you forget to include data for one of the times:

```
> df=data.frame(time=c(1980:1985),effort=c(10,5,2,8,1,2))
> dipper.ddl$p=merge_design.covariates(dipper.ddl$p,df)

  Error in merge_design.covariates(dipper.ddl$p,df) :

  df does not contain a time value for each time in design data
```

The dataframe can contain any number of covariates with any valid names and the only restriction is that it must contain a field named **time** with values that match those in the design data. The dataframe can be created as above or functions like **read.table** can be used to import data in a file into a dataframe. For more details on these functions, refer to the **R** help files on **read.table** and **data.frame**.

Let's create another model that uses those new design data.

```
> Phi.ageclass.plus.sex=list(formula=~ageclass+sex)
> p.effort.plus.sex=list(formula=~effort+sex)
> dipper.phi.ageclassplussex.p.effortplussex =
    mark(dipper.process,dipper.ddl,model.parameters=list(Phi=
       Phi.ageclass.plus.sex,p= p.effort.plus.sex))

 Output summary for CJS model
 Name : Phi(~ageclass + sex)p(~effort + sex)

 Npar :  7
 -2lnL:  665.1266
 AICc :  679.3946

 Beta
                     estimate        se        lcl       ucl
 Phi:(Intercept)    0.1964602 0.1750104 -0.1465602 0.5394805
 Phi:ageclass[1,3)  0.1033126 0.2258833 -0.3394186 0.5460439
 Phi:ageclass[3,6] -0.1287800 0.4376419 -0.9865582 0.7289982
 Phi:sexMale        0.0306891 0.2046965 -0.3705160 0.4318943
 p:(Intercept)      2.3193847 0.5699104  1.2023604 3.4364090
 p:effort          -0.0901470 0.1098187 -0.3053917 0.1250977
 p:sexMale          0.4862940 0.6626337 -0.8124681 1.7850561

 Real Parameter Phi Group:sexFemale
           1980      1981      1982      1983      1984      1985
 1980 0.5489577 0.5743870 0.5743870 0.5169136 0.5169136 0.5169136
 1981           0.5489577 0.5743870 0.5743870 0.5169136 0.5169136
 1982                     0.5489577 0.5743870 0.5743870 0.5169136
 1983                               0.5489577 0.5743870 0.5743870
 1984                                         0.5489577 0.5743870
 1985                                                   0.5489577

 Group:sexMale
           1980      1981      1982      1983      1984      1985
 1980 0.5565444 0.5818718 0.5818718 0.5245725 0.5245725 0.5245725
 1981           0.5565444 0.5818718 0.5818718 0.5245725 0.5245725
 1982                     0.5565444 0.5818718 0.5818718 0.5245725
 1983                               0.5565444 0.5818718 0.5818718
 1984                                         0.5565444 0.5818718
 1985                                                   0.5565444
```

Notice the diagonal patterns in **Phi** as it relates to the final **ageclass** definition that we used.

It is also possible to assign group-specific and time-specific covariates to the design data with the **merge_design.covariates** function. The following is an example using the dipper data in which effort is sex (group) specific. A dataframe (df) is constructed with the group and time-specific effort values and this is then used in the call to **merge_design.covariates**. See the help file for that function for more details.

```
> df=data.frame(group=c(rep("Female",7),rep("Male",7)),
           time=rep(c(1980:1986),2),effort=c(10,5,2,8,1,2,3,20,10,4,16,2,4,6))
```

```
> df

     group time effort
 1  Female 1980     10
 2  Female 1981      5
 3  Female 1982      2
 4  Female 1983      8
 5  Female 1984      1
 6  Female 1985      2
 7  Female 1986      3
 8    Male 1980     20
 9    Male 1981     10
 10   Male 1982      4
 11   Male 1983     16
 12   Male 1984      2
 13   Male 1985      4
 14   Male 1986      6

> dipper.process=process.data(dipper,group="sex",begin.time=1980)
> dipper.ddl=make.design.data(dipper.process)
> dipper.ddl$p=merge_design.covariates(dipper.ddl$p,df,bygroup=TRUE)

> dipper.ddl$p

     group cohort age time Cohort Age Time    sex effort
 1  Female   1980   1 1981      0   1    0 Female      5
 2  Female   1980   2 1982      0   2    1 Female      2
 3  Female   1980   3 1983      0   3    2 Female      8
 <...>
 29   Male   1981   2 1983      1   2    2   Male     16
 30   Male   1981   3 1984      1   3    3   Male      2
 31   Male   1981   4 1985      1   4    4   Male      4
 32   Male   1981   5 1986      1   5    5   Male      6
 <...>
 40   Male   1984   1 1985      4   1    4   Male      4
 41   Male   1984   2 1986      4   2    5   Male      6
 42   Male   1985   1 1986      5   1    5   Male      6
```

C.7. Comparing results from multiple models

We have put together quite a few models with lots of different names! So how do we keep track of the models and how do we summarize them for model selection and possible model averaging of parameter estimates? Later we will explain more organized approaches but they all tie back to the functions we will use now. The first function is **`collect.models`** which collects all of the models that have been run into a single list and it calls the function **`model.table`**, although the latter can be called separately. Although **`collect.models`** does have arguments in most cases it will be called without arguments and assigned to a list that you can name to help you remember its contents:

```
> dipper.cjs.results=collect.models()
```

What did this do? It looked through all of the objects in the workspace and collected any object that had a class of "**`mark`**". If the workspace included more than one type of **MARK** model, like "**`CJS`**" and

"**POPAN**", it would have issued a warning message. Although it does not matter for this session, the collection of objects can be limited to a particular type of model as follows:

```
> dipper.cjs.results=collect.models(type="CJS")
```

Like the models which have a class of "**mark**" the list resulting from **collect.models** has a class of "**marklist**" and some of the generic functions treat it differently. For example, the **print** function provides a listing of the **model.table** element of the list rather than printing each list element which are the various model results.

```
> dipper.cjs.results
                                   model npar     AICc DeltaAICc       weight Deviance
3                       Phi(~Flood)p(~1)    3 666.1597   0.00000 0.5767865187 77.62566
4                   Phi(~Flood)p(~Flood)    4 668.1557   1.99605 0.2126073874 77.58357
2                           Phi(~1)p(~1)    2 670.8660   4.70638 0.7548324523 84.36055
11                          Phi(~1)p(~1)    2 670.8660   4.70638 0.7548324523 58.15788
12                          Phi(~1)p(~1)    2 670.8660   4.70638 0.7548324523 84.36055
5                         Phi(~sex)p(~1)    3 672.7331   6.57343 0.0215582207 84.19909
9                        Phi(~time)p(~1)    7 673.9980   7.83838 0.0114533494 77.25297
6                Phi(~sex)p(~Time + sex)    5 674.3101   8.15044 0.0097987244 81.69012
7                   Phi(~sex + age)p(~1)    8 678.9925  12.83286 0.0009427448 80.17008
8                Phi(~sex + age)p(~Time)    9 679.1198  12.96015 0.0008846140 78.21000
1   Phi(~ageclass + sex)p(~effort + sex)    7 679.3946  13.23491 0.0007710649 82.64951
10                    Phi(~time)p(~time)   11 679.5879  13.42824 0.0007000190 74.47310
```

The table of model results is fashioned along the lines of the results table shown in the **MARK** interface. By default the table is displayed in ascending order for AIC_c. The number on the left hand-side of the table is the order of the model in the list. If we look at the names of the list elements we see that the first 12 are the names of the models that we created and the last is the **model.table** which is the dataframe that is displayed above.

```
> names(dipper.cjs.results)
 [1] "dipper.phi.ageclassplussex.p.effortplussex"
 [2] "dipper.phi.dot.p.dot"
 [3] "dipper.phi.flood.p.dot"
 [4] "dipper.phi.flood.p.flood"
 [5] "dipper.phi.sex.p.dot"
 [6] "dipper.phi.sex.p.Timeplussex"
 [7] "dipper.phi.sexplusage.p.dot"
 [8] "dipper.phi.sexplusage.p.Time"
 [9] "dipper.phi.time.p.dot"
[10] "dipper.phi.time.p.time"
[11] "myexample"
[12] "myexample2"
[13] "model.table"
```

The model with the lowest AIC_c is the third model in the list. Notice that models 2, 11 and 12 are all the same model. That is because the collection includes the first examples we created named **myexample** and **myexample2**. We certainly don't want models duplicated in the list and especially if we use model averaging. There are different ways they can be removed from the list. One approach would be to use the **rm** function in **R** to remove them from the workspace and then recreate the list. The more direct approach would be to use the function **remove.mark** to remove models 11 and 12 as follows:

```
> dipper.cjs.results=remove.mark(dipper.cjs.results,c(11,12))
> dipper.cjs.results
                                model npar     AICc DeltaAICc       weight Deviance
3                     Phi(~Flood)p(~1)    3 666.1597   0.00000 0.6478308242 77.62566
4                 Phi(~Flood)p(~Flood)    4 668.1557   1.99605 0.2387947959 77.58357
2                         Phi(~1)p(~1)    2 670.8660   4.70638 0.0615863089 84.36055
5                       Phi(~sex)p(~1)    3 672.7331   6.57343 0.0242136031 84.19909
9                    Phi(~time)p(~1)      7 673.9980   7.83838 0.0128640885 77.25297
6              Phi(~sex)p(~Time + sex)    5 674.3101   8.15044 0.0110056589 81.69012
7                 Phi(~sex + age)p(~1)    8 678.9925  12.83286 0.0010588651 80.17008
8              Phi(~sex + age)p(~Time)    9 679.1198  12.96015 0.0009935742 78.21000
1  Phi(~ageclass + sex)p(~effort + sex)    7 679.3946  13.23491 0.0008660389 82.64951
10                   Phi(~time)p(~time)   11 679.5879  13.42824 0.0007862422 74.47310
```

Each of the 10 models is stored in the list and the individual named objects in the workspace are no longer needed. The names of the model objects can be collected with the function **`collect.model.names`** and easily removed as follows:

```
> rm(list=collect.model.names(ls())) # result of function used as argument to 'rm'
> ls()
 [1] "df"                     "dipper"                 "dipper.cjs.results"
 [4] "dipper.ddl"             "dipper.process"         "p.dot"
 [7] "p.effort.plus.sex"      "p.Flood"                "p.time"
[10] "p.Time"                 "p.time.fixed"           "p.Timeplussex"
[13] "Phi.ageclass.plus.sex" "Phi.dot"                "Phi.Flood"
[16] "Phi.sex"                "Phi.sex.plus.age"       "Phi.sexplusage"
[19] "Phi.time"
```

The objects defined for the parameter model specifications (e.g., **`p.flood`**) remain but the model results were removed from the workspace. You can summarize, print, and manipulate any of the models by simply referring to the model as a particular list element (e.g., **`summary(dipper.cjs.results[[3]])`**). Maintaining the model results in a **`marklist`** is a much tidier way to organize results of analyses; however, more importantly, model averaging requires the results to be contained in a **`marklist`**. Also, adjusting model selection for over-dispersion is much easier if the models are maintained in a **`marklist`**.

C.8. Producing model-averaged parameter estimates

The function **`model.average`** provides model averaging of the real parameters either for a single type of parameter (e.g., "**`Phi`**" or "**`p`**") or for all parameters. No facility is provided for model averaging the beta parameters although all of the values are available in the **`marklist`** to do so. All of the real parameters can be averaged over the models as follows:

```
> dipper.mod.avg=model.average(dipper.cjs.results,vcv=TRUE)
```

By default, the function returns a dataframe of the model averaged estimates with standard errors but not confidence intervals. If you include **`vcv=TRUE`**, it will return a list with a dataframe named estimates which includes the estimates with standard errors and confidence intervals and a variance-covariance matrix.

```
> names(dipper.mod.avg)
[1] "estimates" "vcv.real"
```

Model-averaged estimates, standard errors and confidence intervals are provided in the estimates dataframe:

```
> summary(dipper.mod.avg$estimates)
  par.index        estimate            se                lcl               ucl
Min.   : 1.00   Min.   :0.4771   Min.   :0.02991   Min.   :0.3833   Min.   :0.5719
1st Qu.:21.75   1st Qu.:0.6023   1st Qu.:0.03016   1st Qu.:0.5339   1st Qu.:0.6667
Median :42.50   Median :0.7506   Median :0.03357   Median :0.6609   Median :0.8066
Mean   :42.50   Mean   :0.7364   Mean   :0.03439   Mean   :0.6592   Mean   :0.7956
3rd Qu.:63.25   3rd Qu.:0.9000   3rd Qu.:0.03433   3rd Qu.:0.8237   3rd Qu.:0.9454
Max.   :84.00   Max.   :0.9034   Max.   :0.04926   Max.   :0.8241   Max.   :0.9593
```

The field **par.index** is the parameter index for the **all-different** PIM. In this case the first 42 (2 groups of 21) are for **Phi** and the last 42 are for **p**. Unless you need a covariance between parameters of different types, it is more useful to construct the model-averaged estimates by parameter type because the default design data are added to the estimates dataframe which provides some context for the estimates.

```
> dipper.Phi.mod.avg=model.average(dipper.cjs.results,"Phi",vcv=TRUE)
> dipper.Phi.mod.avg$estimates[1:5,]
  par.index  estimate         se       lcl       ucl fixed  group cohort age time Cohort Age Time    sex
1         1 0.6024905 0.03488516 0.5325433 0.6684866       Female   1980   0 1980      0   0    0 Female
2         2 0.4771467 0.04853963 0.3839480 0.5719644       Female   1980   1 1981      0   1    1 Female
3         3 0.4776554 0.04857463 0.3843736 0.5725222       Female   1980   2 1982      0   2    2 Female
4         4 0.6023430 0.03432131 0.5335474 0.6673187       Female   1980   3 1983      0   3    3 Female
5         5 0.6022495 0.03435730 0.5333826 0.6672924       Female   1980   4 1984      0   4    4 Female
```

The estimates, standard errors and variance-covariance matrix are constructed as described by Burnham and Anderson (2002: chapter 4). Confidence intervals for the model-averaged estimates were somewhat more challenging. To provide valid intervals for bounded parameters (e.g., $0 < \varphi < 1$), the model-average variance-covariance matrix of the real parameters are transformed to a variance-covariance matrix for the estimates transformed into the appropriate link space using the Delta-method (see Appendix B). Then asymptotic 95% normal confidence intervals are constructed for the transformed link values and the interval end points are then back-transformed into real parameters. That same method is used to construct confidence intervals for the real parameters for a single model in **MARK**.

C.9. Quasi-likelihood adjustment

An estimate of $\hat{c}$ for over-dispersion can be derived using the **TEST2+TEST3** χ^2/df from program **RELEASE** (see Chapter 5 for full details).

Program **RELEASE** can be run with the function **release.gof** as shown below with the dipper data:

```
> data(dipper)
> dipper.processed=process.data(dipper,model="CJS",groups="sex")
> release.gof(dipper.processed)

 RELEASE NORMAL TERMINATION
       Chi.square df      P
 TEST2     7.5342  6 0.2743
 TEST3    10.7735 15 0.7685
 Total    18.3077 21 0.6295
```

If you add the argument **view=TRUE**, the **RELEASE** output file named (`Releasennn.out`) will be displayed in a window.

Alternatively $\hat{c}$ can be estimated using the median $\hat{c}$ procedure but this is not currently supported in **RMark**. However, you can export the input file and the output from the global model to **MARK** and use the **MARK** interface to run the median $\hat{c}$ procedure. See section C.21 for a description of exporting to **MARK**.

Adjustments for over-dispersion are implemented with the function **adjust.chat** which sets the value of chat for an individual model or all models in a **marklist**. For example, we will set the value of $\hat{c}$ to 2 for the set of dipper results we just created:

```
> dipper.cjs.results=adjust.chat(2,dipper.cjs.results)
```

Doing so does nothing more than setting an element called $\hat{c}$ in each model to 2 in this case. It does not adjust standard errors or confidence intervals in any of the model objects but that is done with functions that extract the results (e.g., **get.real**). However, it does adjust the **model.table** values:

```
>  dipper.cjs.results
                                    model npar    QAICc DeltaQAICc       weight QDeviance chat
 3                      Phi(~Flood)p(~1)    3 336.1083   0.000000 0.4661602174  38.81283    2
 2                          Phi(~1)p(~1)    2 337.4472   1.338942 0.2386644310  42.18028    2
 4                  Phi(~Flood)p(~Flood)    4 338.1254   2.017100 0.1700307784  38.79179    2
 5                       Phi(~sex)p(~1)    3 339.3950   3.286715 0.0901226832  42.09955    2
 6               Phi(~sex)p(~Time + sex)    5 342.2265   6.118215 0.0218766933  40.84506    2
 9                      Phi(~time)p(~1)    7 344.1330   8.024731 0.0084330973  38.62649    2
1  Phi(~ageclass + sex)p(~effort + sex)    7 346.8313  10.722996 0.0021880957  41.32475    2
 7                  Phi(~sex + age)p(~1)    8 347.6689  11.560662 0.0014393600  40.08504    2
 8               Phi(~sex + age)p(~Time)    9 348.7762  12.667990 0.0008274011  39.10500    2
 10                   Phi(~time)p(~time)   11 351.1128  15.004529 0.0002572427  37.23655    2
```

The **model.table** now contains QAIC_c values and the remaining computations based on it instead of AIC_c. The ordering of the models is also changed in this case.

C.10. Coping with identifiability

Now let's look at the summary output from the **Phi(~time)p(~time)** model which we know will be over-parameterized because only the product of the last φ and p are estimable:

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

Npar :  12  (unadjusted=11)
-2lnL:  656.9502
AICc :  681.7057 (unadjusted=679.58789)

Beta
                   estimate          se          lcl         ucl
Phi:(Intercept)  0.9354557   0.7685213   -0.5708461   2.4417575
Phi:time1981    -1.1982745   0.8706688   -2.9047853   0.5082364
Phi:time1982    -1.0228292   0.8049137   -2.6004601   0.5548017
```

```
Phi:time1983    -0.4198589   0.8091476   -2.0057882   1.1660705
Phi:time1984    -0.5360978   0.8031424   -2.1102571   1.0380614
Phi:time1985     0.2481368 244.9012000 -479.7582200 480.2544900
p:(Intercept)    0.8292835   0.7837354   -0.7068380   2.3654050
p:time1982       1.6556230   1.2913788   -0.8754795   4.1867256
p:time1983       1.5220926   1.0729148   -0.5808205   3.6250057
p:time1984       1.3767410   0.9884819   -0.5606835   3.3141654
p:time1985       1.7950894   1.0688773   -0.2999101   3.8900889
p:time1986      -0.0147563 187.0364400 -366.6061900 366.5766800

Real Parameter Phi Group:sexFemale
          1980      1981      1982      1983      1984      1985
1980 0.7181808 0.4346709 0.4781705 0.6261176 0.5985334 0.7655931
1981           0.4346709 0.4781705 0.6261176 0.5985334 0.7655931
1982                     0.4781705 0.6261176 0.5985334 0.7655931
1983                               0.6261176 0.5985334 0.7655931
1984                                         0.5985334 0.7655931
1985                                                   0.7655931

Group:sexMale
          1980      1981      1982      1983      1984      1985
1980 0.7181808 0.4346709 0.4781705 0.6261176 0.5985334 0.7655931
1981           0.4346709 0.4781705 0.6261176 0.5985334 0.7655931
1982                     0.4781705 0.6261176 0.5985334 0.7655931
1983                               0.6261176 0.5985334 0.7655931
1984                                         0.5985334 0.7655931
1985                                                   0.7655931

Real Parameter p Group:sexFemale
          1981      1982      1983      1984      1985      1986
1980 0.6962034 0.9230769 0.9130435 0.9007892 0.9324138 0.6930734
1981           0.9230769 0.9130435 0.9007892 0.9324138 0.6930734
1982                     0.9130435 0.9007892 0.9324138 0.6930734
1983                               0.9007892 0.9324138 0.6930734
1984                                         0.9324138 0.6930734
1985                                                   0.6930734

Group:sexMale
          1981      1982      1983      1984      1985      1986
1980 0.6962034 0.9230769 0.9130435 0.9007892 0.9324138 0.6930734
1981           0.9230769 0.9130435 0.9007892 0.9324138 0.6930734
1982                     0.9130435 0.9007892 0.9324138 0.6930734
1983                               0.9007892 0.9324138 0.6930734
1984                                         0.9324138 0.6930734
1985                                                   0.6930734
```

Note that the number of parameters is shown as 12 and AIC_c is calculated based on 12, but an unadjusted parameter count and AIC_c are also shown with the proper count of 11. The **mark** function assumes that all parameters are identifiable and if the parameter count in the **MARK** output is less than the number of columns in the design matrix, it adjusts the count and AIC_c value if the

default value of the argument **adjust=TRUE** is used. It also keeps the values reported by **MARK** in **results$npar.unadjusted** and **results$AICc.unadjusted** and these are reported in **summary**.

Why not trust the values computed by **MARK**? The ability of **MARK** to count the number of parameters correctly is impaired when using design matrices and it will often not count parameters that are estimable but are at a boundary (0 or 1 for φ or p) which can happen easily with sparse data sets (the technical details of how **MARK** counts parameters are presented in Chapter 4). Overly complex models that have numerous parameters that are at boundaries can appear to be the best model because the parameters are counted improperly. It is more conservative to assume that all parameters are estimable.

When you know that some parameters are not identifiable and should not be counted there are a couple of ways to proceed. One approach is to fix the value of one of the parameters to 1 so it will not be counted and the other parameter is then an estimate of the product of the parameters. This can be done with the argument fixed in the parameter specification list as follows:

```
p.time.fixed=list(formula=~time,fixed=list(time=1986,value=1))
dipper.phi.time.p.time=
    mark(dipper.process,dipper.ddl,model.parameters=list(Phi=Phi.time,p=p.time.fixed))

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

Npar :  11 -2lnL:  656.9502 AICc :  679.5879

Beta
                  estimate        se        lcl       ucl
Phi:(Intercept)  0.9354601 0.7685246 -0.5708483 2.4417684
Phi:time1981    -1.1982793 0.8706724 -2.9047973 0.5082387
Phi:time1982    -1.0228337 0.8049168 -2.6004706 0.5548031
Phi:time1983    -0.4198627 0.8091504 -2.0057975 1.1660720
Phi:time1984    -0.5361021 0.8031460 -2.1102683 1.0380640
Phi:time1985    -0.8128580 0.7947326 -2.3705340 0.7448179
p:(Intercept)    0.8292792 0.7837366 -0.7068447 2.3654031
p:time1982       1.6556296 1.2913815 -0.8754783 4.1867374
p:time1983       1.5220968 1.0729155 -0.5808177 3.6250112
p:time1984       1.3767444 0.9884827 -0.5606817 3.3141704
p:time1985       1.7950930 1.0688789 -0.2999097 3.8900957
p:time1986       0.0000000 0.0000000  0.0000000 0.0000000

Real Parameter Phi Group:sexFemale
          1980      1981      1982      1983      1984      1985
1980 0.7181817 0.4346708 0.4781705 0.6261177 0.5985334 0.5306122
1981           0.4346708 0.4781705 0.6261177 0.5985334 0.5306122
1982                     0.4781705 0.6261177 0.5985334 0.5306122
1983                               0.6261177 0.5985334 0.5306122
1984                                         0.5985334 0.5306122
1985                                                   0.5306122

Real Parameter p Group:sexFemale
          1981      1982      1983      1984      1985 1986
1980 0.6962025 0.9230771 0.9130435 0.9007891 0.9324138    1
1981           0.9230771 0.9130435 0.9007891 0.9324138    1
1982                     0.9130435 0.9007891 0.9324138    1
1983                               0.9007891 0.9324138    1
1984                                         0.9324138    1
```

```
1985                                                    1
```

Fixing parameters can get a little tricky with additive models, so an alternative approach is to use **adjust=FALSE** with **mark** to accept the parameter counts from **MARK** or afterward you can use the function **adjust.parameter.count** to change the parameter count to a new value and AIC_c is subsequently recalculated. If you are going to accept the **MARK** parameter counts, make sure they are correct! In complex models with dozens of parameters, it is quite possible that the optimization code does not reach the global maximum and parameters end up at boundaries and are not counted. Indices for the parameters that are not counted by **MARK** are stored in **results$singular**. You should always check these parameters and ascertain whether it is likely that they are at boundaries and whether they are estimable. If you have any doubts, rerun the model with new starting values as we show in the next example.

The final example we ran earlier demonstrates a situation in which a parameter is at a boundary but is properly estimated:

```
> summary(dipper.phi.sexplusage.p.dot)

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

Npar :  8  (unadjusted=7)
-2lnL:  662.6472
AICc :  678.9925 (unadjusted=676.91513)

Beta
                    estimate           se         lcl         ucl
Phi:(Intercept)   0.1647608 1.696575e-01  -0.1677680   0.4972896
Phi:sexMale       0.0830684 1.995167e-01  -0.3079844   0.4741211
Phi:age1          0.0173059 2.538808e-01  -0.4803006   0.5149123
Phi:age2          0.3599325 3.692076e-01  -0.3637144   1.0835793
Phi:age3         -0.0402832 5.407864e-01  -1.1002246   1.0196581
Phi:age4          0.2645044 8.873705e-01  -1.4747419   2.0037506
Phi:age5        -19.8742890 1.076391e-08 -19.8742890 -19.8742890
p:(Intercept)     2.2565572 3.289010e-01   1.6119113   2.9012031


Real Parameter Phi Group:sexFemale
          1980      1981      1982      1983      1984         1985
1980 0.5410973 0.5453913 0.6282445 0.5310793 0.6056982 2.755882e-09
1981           0.5410973 0.5453913 0.6282445 0.5310793 6.056982e-01
1982                     0.5410973 0.5453913 0.6282445 5.310793e-01
1983                               0.5410973 0.5453913 6.282445e-01
1984                                         0.5410973 5.453913e-01
1985                                                   5.410973e-01
```

```
Group:sexMale
          1980      1981      1982      1983      1984         1985
1980 0.5616421 0.5658982 0.6474300 0.5517010 0.6253534 2.994585e-09
1981           0.5616421 0.5658982 0.6474300 0.5517010 6.253534e-01
```

```
1982                     0.5616421 0.5658982 0.6474300 5.517010e-01
1983                               0.5616421 0.5658982 6.474300e-01
1984                                         0.5616421 5.658982e-01
1985                                                   5.616421e-01


Real Parameter p Group:sexFemale
          1981      1982      1983      1984      1985      1986
1980 0.9052146 0.9052146 0.9052146 0.9052146 0.9052146 0.9052146
1981           0.9052146 0.9052146 0.9052146 0.9052146 0.9052146
1982                     0.9052146 0.9052146 0.9052146 0.9052146
1983                               0.9052146 0.9052146 0.9052146
1984                                         0.9052146 0.9052146
1985                                                   0.9052146

Group:sexMale
          1981      1982      1983      1984      1985      1986
1980 0.9052146 0.9052146 0.9052146 0.9052146 0.9052146 0.9052146
1981           0.9052146 0.9052146 0.9052146 0.9052146 0.9052146
1982                     0.9052146 0.9052146 0.9052146 0.9052146
1983                               0.9052146 0.9052146 0.9052146
1984                                         0.9052146 0.9052146
1985                                                   0.9052146


> dipper.phi.sexplusage.p.dot$results$singular
[1] 7
```

The seventh β for the age 5 φ effect is at a boundary such that survival from age 5 to 6 is estimated to be zero, We can see if this a numerical problem by rerunning the model and changing the initial value for beta 7 using the **initial** argument of **mark** as follows:

```
> initial= dipper.phi.sexplusage.p.dot$results$beta$estimate
> initial[7]=0
> dipper.phi.sexplusage.p.dot
    =mark(dipper.process,dipper.ddl,model.parameters
      =list(Phi=Phi.sexplusage,p=p.dot),initial=initial)
```

Setting the "singular" βs to zero and refitting the model will often help the optimization move away from the boundary and find the global maximum. That is the approach that is taken if you set the argument **retry** in **mark** to a non-zero value. Upon fitting a model and finding singular β values, it will refit the model the specified number of times, using the initial values from the previous fitting but setting the initial value of singular βs to 0. However, in this case, re-running the analysis produces the same result. A quick check of the capture histories for the first release cohort shows that there was not a single encounter of the first cohort on the last occasion:

```
> dipper$ch[substr(dipper$ch,1,1)==1]
 [1] "1000000" "1000000" "1000000" "1000000" "1000000" "1000000" "1000000"
 [8] "1000000" "1000000" "1010000" "1010000" "1100000" "1100000" "1100000"
[15] "1100000" "1100000" "1100000" "1101110" "1111000" "1111000" "1111100"
[22] "1111110"
```

Thus, with an assumed constant capture probability the best explanation of not seeing any on the seventh occasion is that survival from age 5 to 6 was 0. The parameter is identifiable and is being estimated correctly but it is at a boundary and is not being counted correctly by **MARK**. Moral of the story is to be careful counting parameters (this point has been made at several points in this book). The philosophy incorporated into **RMark** is that it is safer to over-count parameters rather than risk fitting an overly-complex model to sparse data.

The ability of **MARK** to count parameters can be improved by using the sin link which is now supported by **RMark** as long as the resulting design matrix for the parameter is an identity matrix. The sin link can be used for some parameters and a different link for others, so the entire design matrix need not be an identity matrix. If the model formula contains any design or individual covariates then the sin link is not allowed. Also, to use the sin link the formula cannot be additive or use an intercept. If either of the above occurs, an error message will be generated if you specify the sin link. To specify an identity design matrix there must be a 1:1 relationship between the β's and the real parameters. Because **RMark** simplifies the PIMS this can occur even when the group structure is quite complex.

As an example, we will use **`example.data`** which has sex, age and region factors for grouping. Even though there are many parameters in the all-different formulation we can use the sin link with the intercept model (as shown below) because there is one β and one real.

```
> data(example.data)
> example.processed=process.data(example.data,groups=c("sex","age","region"),initial.ages=c(0,1,2))
> example.ddl=make.design.data(example.processed)
> mark(example.processed,example.ddl,model.parameters=list(Phi=list(formula=~1,link="sin")),output=F)
```

For the ~**`time`** model there is also one β for each real parameter but if we specify the model with the sin link, we get an error message:

```
> mark(example.processed,example.ddl,
      model.parameters=list(Phi=list(formula=~time,link="sin")),output=F)

 Error in make.mark.model(data.proc, title = title, covariates = covariates,  :

 Cannot use sin link with non-identity design matrix
```

The error occurs because ~**`time`** creates a design matrix with an intercept and a β for each time beyond the first time, so it is *additive* which is not allowed. However, we can specify a design matrix without an intercept using ~**`-1 + time`** as shown below, or ~**`-1+sex`**:

```
> mark(example.processed,example.ddl,model.parameters=list(Phi=list(formula=~-1+time,link="sin")),output=F)
> mark(example.processed,example.ddl,model.parameters=list(Phi=list(formula=~-1+sex,link="sin")),output=F)
```

Likewise, we can use the sin link with full interaction models as long as the intercept is removed.

```
> mark(example.processed,example.ddl,model.parameters=
   list(Phi=list(formula=~-1+region:time,link="sin")),output=F)
```

But again you cannot have any additive terms even in the case of adding 2 two-way interactions:

```
> mark(example.processed,example.ddl,model.parameters=
      list(Phi=list(formula=~-1+region:time+sex:time,link="sin")),output=F)

 Error in make.mark.model(data.proc, title = title, covariates = covariates,  :

 Cannot use sin link with non-identity design matrix
```

But the 3-way interaction model can use the sin link:

```
> mark(example.processed,example.ddl,model.parameters=
      list(Phi=list(formula=~-1+sex:region:time,link="sin")),output=F)
```

C.11. Fixing real parameter values

Parameter confounding presented a situation in which it was useful to fix specific real parameter values and in C.10 we showed how that could be done with the **fixed** argument of the parameter specification list. However, there are other instances in which real parameter values need to be fixed and there are several ways in which fixed parameters can be specified with **RMark**. In addition to parameter confounding, real parameters are typically fixed in circumstances in which there are no data to estimate the parameter (i.e., a structural zero). For example, imagine a scenario where you conducted a "CJS" study in which new animals were only released every other year. In those years with no releases there cannot be any recaptures from that cohort to estimate the parameters. For the limiting case in which only one cohort is released and then followed through time, see discussion about **pim.type** at the end of this section. Another example would be a Multistrata model in which the strata are defined such that some transitions are not possible, so they would be fixed to 0.

There are 2 general approaches to specification of fixed parameters. The first approach was introduced in C.10 using the **fixed** argument which identifies specific parameters by their indices and specifies their fixed values. The second approach is to delete the rows from the design data for the parameters that are to be fixed at a single default value for each type of parameter (e.g., φ or p). If you need to fix parameters of the same type to different values (e.g., some $p = 0$ and others $p = 1$), you need to use the first approach. The second approach is most useful when all the parameters are being fixed to the same value because of missing data (i.e., structural zero). We will use the dipper data to illustrate how to fix real parameters using these different approaches.

There are 4 different forms for the fixed argument. The first sets all of the parameters of a particular type to the same value. For example, the following poor and non-realistic model would set all of the φ values to 1 for the dipper data.

```
> dipper.processed=process.data(dipper,groups=("sex"),begin.time=1980)
> dipper.ddl=make.design.data(dipper.processed)
> dipper.ddl$p
> Phidot=list(formula=~1)
> Phi.1=list(formula=~1,fixed=1)
> mark(dipper.processed,dipper.ddl,model.parameters=list(Phi=Phi.1))

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

 Npar :  1
 -2lnL:  981.2354
 AICc :  983.2449

 Beta
                  estimate        se       lcl        ucl
 Phi:(Intercept)  0.000000 0.0000000  0.000000  0.0000000
 p:(Intercept)   -1.018446 0.0777791 -1.170893 -0.8659991
```

```
Real Parameter Phi
Group:sexFemale
     1980 1981 1982 1983 1984 1985
1980    1    1    1    1    1    1
1981         1    1    1    1    1
1982              1    1    1    1
1983                   1    1    1
1984                        1    1
1985                             1

Group:sexMale
     1980 1981 1982 1983 1984 1985
1980    1    1    1    1    1    1
1981         1    1    1    1    1
1982              1    1    1    1
1983                   1    1    1
1984                        1    1
1985                             1
```

Fixing all of the parameters to one value is most useful to simplify the model structure. For example, setting the fidelity parameter (F) in the Burnham model for the case in which the recovery and recapture areas are the same or setting the resight probability (**R** and **RPrime**) in the Barker model to zero to get the Burnham model.

The other forms of the fixed argument involve specifying a set of times, cohorts, ages, groups or generic indices and a set of one or more values. The first 4 are simply short-cuts for the most general approach of specifying indices. Let's start with a generalization of the approach given in C.10 in which we want to fix $p = 0$ in 1982 and 1984 (presumably because of no sampling):

```
> p.time.fixed=list(formula=~time,fixed=list(time=c(1982,1984),value=0))
> mark(dipper.processed,dipper.ddl,model.parameters=list(p=p.time.fixed))

Real Parameter p
Group:sexFemale
          1981 1982      1983 1984      1985      1986
1980 0.9343357    0 0.5387934    0 0.8816249 0.9999979
1981              0 0.5387934    0 0.8816249 0.9999979
1982                0.5387934    0 0.8816249 0.9999979
1983                             0 0.8816249 0.9999979
1984                               0.8816249 0.9999979
1985                                         0.9999979

Group:sexMale
          1981 1982      1983 1984      1985      1986
1980 0.9343357    0 0.5387934    0 0.8816249 0.9999979
1981              0 0.5387934    0 0.8816249 0.9999979
1982                0.5387934    0 0.8816249 0.9999979
1983                             0 0.8816249 0.9999979
1984                               0.8816249 0.9999979
1985                                         0.9999979
```

The same approach will work if you specify certain age, cohort or group values. The use of group is restricted to the group numbers and not the factor variables defining the groups.

Now you would think that the following would work to constrain p for 1982 to 0 and p for 1986 to 1, but it does not (although the programming could be changed) because it expects to have as many values as there are parameters associated with times 1982 and 1986.

```
> p.time.fixed=list(formula=~time,fixed=list(time=c(1982,1986),value=c(0,1)))
> mark(dipper.processed,dipper.ddl,model.parameters=list(p=p.time.fixed))

Lengths of indices and values do not match for fixed parameters for p
Error in make.mark.model(data.proc, title = title, covariates = covariates,
```

That brings us to the final approach which is to specify the parameter indices and the values for those parameters. The indices are the row numbers of the design data for the parameter. For example, in the first 10 rows of the p design data, the indices for 1982 are 2 and 7:

```
  dipper.ddl$p[1:10,]
     group cohort age time Cohort Age Time    sex
 1  Female   1980   1 1981      0   1    0 Female
 2  Female   1980   2 1982      0   2    1 Female
 3  Female   1980   3 1983      0   3    2 Female
 4  Female   1980   4 1984      0   4    3 Female
 5  Female   1980   5 1985      0   5    4 Female
 6  Female   1980   6 1986      0   6    5 Female
 7  Female   1981   1 1982      1   1    1 Female
 8  Female   1981   2 1983      1   2    2 Female
 9  Female   1981   3 1984      1   3    3 Female
 10 Female   1981   4 1985      1   4    4 Female
```

Now you certainly don't want to look them up and type them in because you will almost certainly make a mistake and it would disable automatic updating of the model if the group structure changed or another occasion was added. The solution is to use a little **R** code to define the set of indices as follows:

```
> p1982.indices=as.numeric(row.names(dipper.ddl$p[dipper.ddl$p$time==1982,]))
> p1982.indices
 [1]  2  7 23 28
> p1986.indices=as.numeric(row.names(dipper.ddl$p[dipper.ddl$p$time==1986,]))
> p1986.indices
 [1]  6 11 15 18 20 21 27 32 36 39 41 42
```

The above code selects the indices which have a time of 1982 and stores it into **`p1982.indices`** and likewise for 1986. That code will work even if the group or data structure changes as long as the **`begin.time`** doesn't change but even that part of the code could be automated. Now you don't want to count how many values need to be set to 0 and 1, so again we use some **R** code to do the work:

```
>  p1982.values=rep(0,length(p1982.indices))
>  p1986.values=rep(1,length(p1986.indices))
>  p1982.values
 [1] 0 0 0 0
> p1986.values
  [1] 1 1 1 1 1 1 1 1 1 1 1 1
```

Finally, you can put it all together as follows:

```
> p.time.fixed=list(formula=~time,fixed=list(index=c(p1982.indices,p1986.indices),
                                  value=c(p1982.values,p1986.values)))
> mark(dipper.processed,dipper.ddl,model.parameters=list(p=p.time.fixed))

 Real Parameter p
 Group:sexFemale
           1981 1982      1983      1984     1985 1986
 1980 0.9720207    0 0.8387273 0.8880947 0.938504    1
 1981              0 0.8387273 0.8880947 0.938504    1
 1982                0.8387273 0.8880947 0.938504    1
 1983                          0.8880947 0.938504    1
 1984                                    0.938504    1
 1985                                                1

 Group:sexMale
           1981 1982      1983      1984     1985 1986
 1980 0.9720207    0 0.8387273 0.8880947 0.938504    1
 1981              0 0.8387273 0.8880947 0.938504    1
 1982                0.8387273 0.8880947 0.938504    1
 1983                          0.8880947 0.938504    1
 1984                                    0.938504    1
 1985                                                1
```

It may help to examine the value for **fixed**, which we can see is a list with the 2 sets of indices (**$index**) pasted (concatenated) together and a set of values (**$value**), which contain 4 zeros for the 1982 parameters and 12 ones for the 1986 parameters.

```
> list(index=c(p1982.indices,p1986.indices),value=c(p1982.values,p1986.values))

  $index
  [1]  2  7 23 28  6 11 15 18 20 21 27 32 36 39 41 42

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

The above approach is completely general and you can use the same pattern and simply change the subset of parameters that are selected. Without showing the results, the following snippet of code could be used to set p in 1982 for females to 0 but for males it sets p in 1984 to 0:

```
> p1982.f=as.numeric(row.names(dipper.ddl$p[dipper.ddl$p$time==1982&
                                        dipper.ddl$p$sex=="Female",]))
> p1984.m=as.numeric(row.names(dipper.ddl$p[dipper.ddl$p$time==1984&
                                        dipper.ddl$p$sex=="Male",]))
> p.time.fixed=list(formula=~time,fixed=list(index=c(p1982.f,p1984.m),value=0))
> mark(dipper.processed,dipper.ddl,model.parameters=list(p=p.time.fixed))
```

Now if the parameters need to be fixed to model structural zeros in the data, then deleting the design data for the parameters representing the missing data is typically the easiest approach. To demonstrate with an example, below we stripped the 1982 cohort from the dipper data and saved it in **xdipper**. After processing the data and making the design data, we deleted the φ and p design data for 1982

by copying all design data other than the data for the 1982 cohort. When the model is run, the default summary shows blanks for parameters with deleted design data. When the model is summarized with the argument **show.fixed=TRUE**, then the default parameter values of $p = 0$ and $\varphi = 1$ are shown for the 1982 cohort.

```
> xdipper=dipper[!substr(dipper$ch,1,3)=="001",]
> xdipper.processed=process.data(xdipper,groups=("sex"),begin.time=1980)
> xdipper.ddl=make.design.data(xdipper.processed)
> xdipper.ddl$p=xdipper.ddl$p[xdipper.ddl$p$cohort!=1982,]
> xdipper.ddl$Phi=xdipper.ddl$Phi[xdipper.ddl$Phi$cohort!=1982,]
> xdipper.model=mark(xdipper.processed,xdipper.ddl)


> summary(xdipper.model,show.fixed=TRUE)

Real Parameter Phi
Group:sexFemale
          1980      1981      1982      1983      1984      1985
1980 0.5886486 0.5886486 0.5886486 0.5886486 0.5886486 0.5886486
1981           0.5886486 0.5886486 0.5886486 0.5886486 0.5886486
1982                     1.0000000 1.0000000 1.0000000 1.0000000
1983                               0.5886486 0.5886486 0.5886486
1984                                         0.5886486 0.5886486
1985                                                   0.5886486

Group:sexMale
          1980      1981      1982      1983      1984      1985
1980 0.5886486 0.5886486 0.5886486 0.5886486 0.5886486 0.5886486
1981           0.5886486 0.5886486 0.5886486 0.5886486 0.5886486
1982                     1.0000000 1.0000000 1.0000000 1.0000000
1983                               0.5886486 0.5886486 0.5886486
1984                                         0.5886486 0.5886486
1985                                                   0.5886486

Real Parameter p
Group:sexFemale
          1981      1982      1983      1984      1985      1986
1980 0.8919246 0.8919246 0.8919246 0.8919246 0.8919246 0.8919246
1981           0.8919246 0.8919246 0.8919246 0.8919246 0.8919246
1982                     0.0000000 0.0000000 0.0000000 0.0000000
1983                               0.8919246 0.8919246 0.8919246
1984                                         0.8919246 0.8919246
1985                                                   0.8919246

Group:sexMale
          1981      1982      1983      1984      1985      1986
1980 0.8919246 0.8919246 0.8919246 0.8919246 0.8919246 0.8919246
1981           0.8919246 0.8919246 0.8919246 0.8919246 0.8919246
1982                     0.0000000 0.0000000 0.0000000 0.0000000
1983                               0.8919246 0.8919246 0.8919246
```

```
1984                                        0.8919246 0.8919246
1985                                                  0.8919246
```

Because structural zeros can be a common occurrence with missing cohorts, a function argument **remove.unused** was added to **make.design.data**. If it is set to **TRUE**, then the design data is automatically deleted for any cohorts without any releases. Thus, the example above can be run with the following commands:

```
> xdipper.ddl=make.design.data(xdipper.processed,remove.unused=TRUE)
> xdipper.model=mark(xdipper.processed,xdipper.ddl)
> summary(xdipper.model,show.fixed=TRUE)
```

Some of the parameters have a natural default value (Table C.1) that is assigned if the design data are deleted but on occasion you may want to change the default value or assign a default value to a parameter that has no assigned default value. That is accomplished by setting the argument default in the parameter specification list. In the following rather silly example, the defaults for the above analysis are set to $p = 0.9$ and $\varphi = 0.5$:

```
> xdipper.model=mark(xdipper.processed,xdipper.ddl,
                     model.parameters=list(p=list(default=.9),Phi=list(default=.5)))
> summary(xdipper.model,show.fixed=TRUE)
```

In some cases, you can handle the structural zeros by using a PIM type other than the default "all different". It can be useful to use **pim.type="time"** or **pim.type="constant"** in the call to **make.design.data**. If you choose one of these simpler PIM structures, you cannot use formula for that parameter that is more complex than the structure allows. A constant PIM can be useful to simplify a model by fixing a real parameter at a value ($F = 1$ for Burnham model) or only allowing models with a single parameter to be estimated. A time PIM can be used in a similar situation for triangular PIMS which can be useful with the CJS model for a single release cohort. Using **pim.type="time"** eliminates the need for deleting the unneeded design data and the summary printout of real parameters is limited to a single line. We demonstrate with the dipper data which is restricted to the data from the first release cohort:

```
> data(dipper)
> dipper=dipper[substr(dipper$ch,1,1)=="1",]
> mark(dipper,design.parameters=list(p=list(pim.type="time"),
      Phi=list(pim.type="time")))

 Real Parameter Phi

           1         2         3         4         5         6
 1 0.6043321 0.6043321 0.6043321 0.6043321 0.6043321 0.6043321


 Real Parameter p

           2         3         4         5         6         7
 1 0.8207724 0.8207724 0.8207724 0.8207724 0.8207724 0.8207724
```

Had we not used **pim.type="time"**, then summary would have shown the entire triangular PIM even if the design data were deleted.

C.12. Data Structure and Import for RMark

So far we have only used the dipper data that accompanies **RMark** and have not discussed data requirements. There are numerous other example datasets to demonstrate other models in **RMark** (e.g., **BlackDuck**, **mallard**, **mstrata** and many others). However, to use **RMark** for your own data you need to understand the requirements for the data structure and how to input data. Data for **RMark** must exist in an **R** dataframe with some formatting and naming conventions. There are numerous ways to create dataframes in **R** but we will describe two functions in **RMark** to help in creating the necessary dataframe.

The format for data input with **RMark** is different than with **MARK**, but a function **convert.inp** was written to convert an **.inp** file used for **MARK** to a dataframe for **RMark**. The conversion is necessary because the data format and structures for **MARK** and **RMark** have a fundamental difference in handling groups, as illustrated below. The formats are similar in that each record (row) contains a capture history which can represent one or more animals as specified by the count (freq) and any number of covariates can be tacked on at the end of the record. However, with **MARK**, group structure is accommodated by having a count (freq) for each group but the data do not contain any information about what was used to construct the groups. The group structure is only represented by group labels. In comparison with **RMark**, each record only represents animals from a single group and the record can contain columns for factor variables that are used to define the group structure.

First we will start with a simple example to show how easy it is to convert an `.INP` file and then we'll work into more complicated examples. The **Examples** subdirectory of **MARK** contains a file **pradel.inp** which can be converted to a dataframe for **RMark** and we can look at the first 5 rows with the following commands:

```
> pradel=convert.inp("C:/Program Files/MARK/Examples/pradel.inp")
 pradel[1:5,]
              ch freq
 1 0000000000001   47
 2 0000000000010   36
 3 0000000000011   12
 4 0000000000100   30
 5 0000000000101    8
```

The first argument value "`C:/Program Files/MARK/Examples/pradel.inp`" is the name of the **MARK** **.inp** file which is shown here with the full path and file specification. Make sure to use a single forward slash or two backslashes to separate sub-directories (note: forward slash? backward slash? The differences reflect the convention used in **R** to accommodate different directory naming conventions between Windows on the one hand, and almost everything else on the other). The extension is assumed to be **.inp** but if it is something different, you can specify the extension with the filename (e.g., "**pradel.txt**").

If there is no group structure or covariates as in the above example then the conversion is quite easy, but it does not get much more complicated. Now let's consider the **MARK** example **Pass3MStrata5.inp** which has 2 covariates weight and length and also uses comments to identify each row uniquely. First we will show what happens if you get it wrong:

```
> p3m5=convert.inp("C:/Program Files/MARK/Examples/Pass3MStrata5.inp")

  Error in convert.inp("C:/Program Files/MARK/Examples/Pass3MStrata5.inp") :
 Number of columns in data file does not match group/covariate specification
```

We forgot to specify the 2 covariates that were in the file, so let's try it again:

```
> p3m5=convert.inp("C:/Program Files/MARK/Examples/Pass3MStrata5.inp",
                       covariates=c("weight","length"))
> p3m5[1:5,]
             ch freq weight length
 1 U00000000000    1   1295    548
 2 00000000000U    1   2653    671
 3 000000D0D000    1   1324    528
 4 0000000000W0    1   1415    570
 5 0000D0000000    1    982    500
```

You can see that it ignored the comments contained on each row of the file. If they are unique and we wanted to use them to label the data, we would change it to:

```
> p3m5=convert.inp("C:/Program Files/MARK/Examples/Pass3MStrata5.inp",
              covariates=c("weight","length"),use.comments=TRUE)
> p3m5[1:5,]
                                     ch freq weight length
 1F1-16C-1054 Upper Green   U00000000000    1   1295    548
 1F1-567-2B3A Upper Green   00000000000U    1   2653    671
 1F1-70D-3E7F Desolation    000000D0D000    1   1324    528
 1F1-770-5307 White         0000000000W0    1   1415    570
 1F1-940-3256 Desolation    0000D0000000    1    982    500
```

Had the comments not provided unique labels then the conversion would have failed and the **use.comments** argument would have to be deleted.

Now let's consider how to convert an **.inp** file that contains a group structure. We will use the **MARK** example file **huggins.inp** which has 2 groups. The example doesn't label those groups but we'll assume that the first group was for females and the second for males. To convert the file, we need to specify a value for the argument **group.df** which is a dataframe that specifies the covariates that will be assigned to each group in the **RMark** dataframe. The rows of **group.df** must correspond to the columns of the frequency field in the **.inp** file. In this simple example, there are 2 columns so there will be 2 rows in our dataframe which will be specified as **group.df=data.frame(sex=c("Female","Male"))**.

Let's dissect that command. First **c("Female","Male")** creates a vector by pasting (concatenating) the strings **''Female''** and **''Male''**. Then the vector is assigned to "**sex**" which will be the name in the **R** dataframe for that group factor value. So the commands to convert and look at the first and last few records are:

```
> huggins=convert.inp("C:/Program Files/MARK/Examples/huggins.inp",
                       group.df=data.frame(sex=c("Female","Male")))
> head(huggins)
          ch freq    sex
 1:1 0101010    1 Female
 1:2 0011000    1 Female
 1:3 1001100    1 Female
 1:4 1100101    1 Female
 1:5 0101010    1 Female
 1:6 1011011    1 Female
```

```
> tail(huggins)
            ch freq  sex
 2:389 0001111    1 Male
 2:390 0010000    1 Male
 2:391 1000101    1 Male
 2:392 0100000    1 Male
 2:393 1010100    1 Male
 2:394 1001010    1 Male
```

Now let's move onto a more complicated group structure. Below is a table showing the first 4 records from the swift dataset (**multi_group.inp** file) that is described in chapter 6. Groups represent 2 levels of sex (female/male) and 2 levels (good/poor) of colony and because there are 4 groups, there are an equivalent number of frequency fields for each capture history. For example there are 145 Females in 'poor' colonies with the capture history **0010** and 171 Males in 'good' colonies with the same capture history. Here are the first 4 records in the file:

history	*female, good*	*female, poor*	*male, good*	*male, poor*
0010	150	145	171	167;
0011	200	205	179	183;
0100	213	198	131	77;
0101	14	26	32	50;

As shown below, the same 4 records expand to 16 records in the **RMark** format because each record corresponds to a single animal or group of animals. If one or more of the frequencies is zero the record is not needed. While the **MARK** format can be more compact it is less flexible in modifying the group structure. The **RMark** data format can be created from the **MARK** format with the function **convert.inp**. The function call for this example would be:

```
multigroup=convert.inp("multi_group",
   group.df=data.frame(sex=c(rep("Female",2),rep("Male",2)),
   Colony=rep(c("Good","Poor"),2)))
```

Here is the format of the **RMark** dataframe for same **multi_group.inp** data file.

history	*frequency*	**sex**	**colony**
0010	150	female	good
0011	200	female	good
0100	213	female	good
0101	14	female	good
0010	145	female	poor
0011	205	female	poor
0100	198	female	poor
0101	26	female	poor
0010	171	male	good
0011	179	male	good
0100	131	male	good
0101	32	male	good
0010	167	male	poor

0011	183	male	poor
0100	77	male	poor
0101	50	male	poor

The function argument **group.df** specifies the factor variables that will be used to define the 4 groups in the **.inp** file. It is a dataframe and it must contain a record for each group in the left to right order they are given in the **.inp** file. In this file the first 2 groups are for females and the last 2 are for males and the colony type alternates (good, poor, good, poor). Let's dissect the value assigned to **group.df**:

```
data.frame(sex=c(rep("Female",2),rep("Male",2)),colony=rep(c("Good","Poor"),2))
```

The call to function **data.frame** creates an **R** dataframe with 2 columns named **sex** and **colony**. The names can be any valid name and the order in the dataframe is not relevant. Had there been more fields they could have been added by assigning more columns in the dataframe. What *does* matter is that the columns are all of the same length and for this particular dataframe the order of the rows must match the order of the columns in the **MARK** **.inp** file.

```
sex=c(rep("Female",2),rep("Male ",2))
```

creates a vector of length 4 that is "Female", "Female", "Male", "Male". The function **rep()** creates a vector by *repeating* the first argument value ("Female" or "Male") the number of times specified as the second argument value (2 in this case). The function **c()** concatenates (pastes) together its arguments in the order they are specified. So it sticks together the 2 vectors each with 2 elements into a single vector of length 4.

The code

```
colony=rep(c("Good","Poor"),2)
```

repeats the vector **c("Good","Poor")** twice to yield a vector with 4 elements "Good","Poor","Good", "Poor" which is the order we want.

The resulting **group.df** looks as follows:

```
  sex      Colony
1 Female   Good
2 Female   Poor
3  Male    Good
4  Male    Poor
```

Notice that the values are not shown in quote marks as they were specified. When columns in a dataframe are specified with character strings they are coerced into factor variables by default and that is what we want in this case. What is stored in the dataframe is actually an index to a factor level (numerically 1 or 2 in this case) and what is shown is the label for the factor. This is apparent if we ask for a summary of the fields (columns) or look at just a single column:

```
> summary(group.df)
     sex      Colony
 Female:2    Good:2
 Male  :2    Poor:2

> group.df$sex
[1] Female Female Male   Male Levels: Female Male.
```

By default, levels of a factor variable are assigned in alphabetical order.

These variables are then assigned to the matching capture history and frequency to create a record in the resulting **RMark** dataframe which has been named **multigroup** in this example. The resulting dataframe **multigroup** would look like the representation of the table above if it were sorted by **sex** and **colony** for the first four records.

If the **MARK .inp** file also contains individual covariates, the names of these are specified in the covariates argument of **convert.inp**. For example, the call to **convert.inp** for the example file **indcov1** from chapter 11 is:

```
indcov1=convert.inp("indcov1",covariates=c("cov1","cov2"))
```

The call specifies that the two covariates should be named **cov1** and **cov2** in the **RMark** dataframe. In this example, there was only a single group, so **group.df** was not specified.

The final argument for **convert.inp** is **use.comments** which can be either **TRUE** or **FALSE** (default). Comments in **MARK .inp** files are given as values between /* and */ and these are ignored in the conversion if they span several lines or the whole line is a comment. However, in some cases the comment is used to specify a label for each capture history (e.g., a tag number for the animal) and it may be useful to retain its value in the **RMark** dataframe. If the values of the comments are unique, you can specify **use.comments=TRUE** and it will use the value for the row name of the dataframe. This is shown using the **blckduck.inp** file that accompanies **MARK**:

```
> bd=convert.inp("blckduck",covariates=c("age","weight","winglen","ci"),
    use.comments=TRUE)
```

The first 5 records of the resulting dataframe show the row names as $01, 04, \cdots, 07$ as follows:

```
> bd[1:5,]
                    ch freq age weight winglen   ci
01  1100000000000000    1   1   1.16    27.7 4.19
04  1011000000000000    1   0   1.16    26.4 4.39
05  1011000000000000    1   1   1.08    26.7 4.04
06  1010000000000000    1   0   1.12    26.2 4.27
07  1010000000000000    1   1   1.14    27.7 4.11
```

If you do not have an existing **MARK .inp** file, an **RMark** dataframe can be constructed from a space or tab-delimited text file using the function **import.chdata**. The function specification is:

```
> import.chdata(filename, header = TRUE, field.names = NULL, field.types = NULL)
```

The argument **filename** specifies the path (if not in directory with workspace) and name (with extension) of a text file. The first field in the file should be the capture history string. It cannot contain any spaces or other delimiting characters and it must be named "**ch**". All other fields can be in any order. The group frequency is not needed if each record specifies the data for a single animal. If frequency is in the file, it should be named "**freq**". The first record in the file can be the names of the fields. If the fields are named, the first field must be named **ch** (for capture history) and if frequency is specified it must be named freq. All other field names for individual covariates can be given any valid name. If the first line contains the field names then the argument header should be specified as the default of **TRUE**. If the first line in the file does not contain the field names, then they should be specified as a vector of strings with the argument field.names. Fields other than "**ch**" should be given a specified field type

unless all fields should be treated as factor variables. The possible field types are "**f**","**n**" and "**s**" for factor, numeric and the last specifies that the field should be skipped and not imported.

The following are function calls to import the text files for 3 of the examples accompanying **RMark**. The files can be found in **Rdata.zip** in **C:\Program Files\R\R-v.vv\library\RMark\data** where *v.vv* is the **R** version number.

```
> example.data<-import.chdata("example.data.txt",field.types=c("n","f","f","f"))}

> edwards.eberhardt<-import.chdata("EdwardsandEberhardt.txt",field.names="ch",
                          header=FALSE)

> dipper<-import.chdata("dipper.txt",field.names=c("ch","sex"),header=FALSE)
```

The first imports **example.data** with 4 fields beyond the capture history. The first field is numeric (**weigh**t) and the last 3 are factor variables (**age**,**sex**,**region**) that are used as grouping variables. The first 6 lines of the file are as follows:

```
ch weight age sex region
1011101  8.3095857 1 M 1
1110000 11.1449917 1 M 2
1000000  4.0252345 1 M 3
1000000  8.6503865 1 M 4
1010000  9.4225103 1 M 1
```

The field names are on the first line of the data file so they are not specified with the **field.names** argument. The next function call is for the Edwards-Eberhardt rabbit data and it has a single field (the capture history) which is not named on the first line, so it is specified with **field.names="ch"** and **header=FALSE**. The last example imports the familiar dipper data which has 2 fields (**capture history** and **sex**) which are specified with the **field.names=c("ch","sex")** and since **sex** is a factor variable, the **field.types** argument need not be specified, but **header=FALSE** is included because the first line does not include field labels.

The above data format and input functions work for most of the models supported by **RMark** with one exception. They do not work with the nest survival model which does not have an encounter history field (**ch**) and requires a much different data format. See the mallard and killdeer datasets for examples of data entry for the nest survival model. For models with an encounter history, the format for **ch** depends on the type of model. For a description of the relevant format see the example(s) provided for each model supported by **RMark** or the model structure description in **MARK**. When the data are processed with the **process.data** function, it checks to make sure that **ch** only contains values that are valid for the chosen analysis model. For example, for **CJS** models, each digit in **ch** can be either a "**0**", "**1**" or "." where "." implies a missing value. In contrast, **ch** for Multistrata models can contain either "**0**" or an alphabetic character which represents the stratum in which it was observed.

C.13. A more organized approach

So far we have introduced **RMark** by typing various commands into **R** and storing the model results in the workspace and then aggregating them into a list. This is a reasonable way to introduce the material but it is not the way we recommend that you conduct your analyses. In this section we will suggest an alternative approach that uses scripts with functions. We recommend this approach for the following reasons:

1. the **R** statements can be stored in a separate text script that can be easily documented
2. the analysis can be easily repeated with the script
3. functions provide an easy way to create a set of models quickly and avoid accidentally aggregating models from different data sets or model types. We will use the swift data set (**aa.inp**) from Chapter 4 as the example.

begin sidebar

Using R functions to ease the workload

R statements can be typed into a text script file and "sourced" into **R** with the "**File | Source R code**" menu selection. **R** expects the script file to have an extension of .R so it is easiest to use it. You can enter and edit the script file with any text editor but you may find it more convenient to work with an editor like TINN-R (**https://sourceforge.net/projects/tinn-r**) which was designed to work with **R**. Such an editor has several advantages including built-in help with **R**, syntax checking and highlighting which helps identify mismatched braces, brackets and parentheses, and automatic sending of scripts or parts of scripts to **R** for execution.

Scripts can contain any valid **R** statements and function calls. So you could simply enter a sequence of statements like we demonstrated in the previous sections. However, we recommend creating a function to define and run the models and then the script contains the definition of the function and the statements to import data and run the function. We recommend using functions because of some of the limitations of **collect.models** and to take advantage of the function **mark.wrapper** that we have not introduced yet. But first we will give a brief description of functions in **R**.

An **R** function is a set of **R** statements that can accept arguments and can return a single value. The **RMark** package is simply a collection of **R** functions with associated help files. All of the built-in **R** functions are in packages but functions can live outside of packages so you can create functions and store them in scripts or in workspaces. So what is the difference between scripts and functions? A *script* is a collection of **R** statements that can be sourced in and they are interpreted in the context of the workspace (**.Rdata**) as if you were typing them at the keyboard. A *function* is subtly different because when a function is executed it effectively creates its own local workspace (called a *frame* in **R**-speak) which contains variables and objects that are defined within the function. The function can use the values of its arguments, the objects it creates and the function can reference any objects from the frame in which it was defined. We will create a simple function that demonstrates this using the **ls()**. A function is composed of a name, an argument list and the body of the function contained in a set of braces ({}). Below we define a very simple function which we call **myls** which has no arguments (no values listed between the parentheses) and the body of the function is a single statement which calls **ls()**:

```
myls=function() { ls() }
```

To illustrate the concept of frames we will call **ls()** and then **myls()** which calls **ls()** within the function.

```
    [1] "aa"                    "aa.models"             "aa.results"
    [4] "df"                    "dipper"                "dipper.cjs.results"
    [7] "dipper.ddl"            "dipper.mod.avg"        "dipper.mod.avg.adj"
   [10] "dipper.Phi.mod.avg"    "dipper.process"        "model.table"
   [13] "myls"                  "p.dot"                 "p.effort.plus.sex"
   [16] "p.Flood"               "p.time"                "p.Time"
   [19] "p.time.fixed"          "p.Timeplussex"         "Phi.ageclass.plus.sex"
   [22] "Phi.dot"               "Phi.Flood"             "Phi.sex"
   [25] "Phi.sex.plus.age"      "Phi.sexplusage"        "Phi.time"
   [28] "summary.mark"
> myls()
character(0)
```

The call to **ls()** shows the contents of the workspace but the call to **ls()** within **myls()** contains no objects because there have been no objects defined within the function. For further illustration we will give **myls()** an argument, define an object and print out some results to show how objects are referenced within functions.

```
myls=function(x) {
                  cat(paste("p.dot = ",p.dot,"\n"))
                  y=x+1
                  p.dot=y
                  cat("p.dot = ",p.dot,"\n") ls()
                 }
```

```
> p.dot
$formula ~1

 > myls(1)
  p.dot =  ~1
  p.dot =  2
  [1] "p.dot" "x"     "y"

 > p.dot
$formula ~1
```

The function was designed to show that **p.dot** from the workspace which we defined earlier could be referenced from within the function but once a value was assigned to **p.dot** within the function it became an object with its own definition within the function that was independent and did not change the p.dot in the workspace. The call to **ls()** within the function shows that there are 3 objects within the local function "workspace" named **x** (the value of the calling argument) and **y** and **p.dot** defined in the function. Functions return the value of the object in the last line of the body of the function (e.g., result of **ls()** in this case) or more specifically a **return()** statement can be used and multiple values can be returned via a list(). Functions can modify objects in the workspace with the use of the ≪− assignment operator but it is not a recommended programming practice.

Because **ls()** only operates on objects defined *within* the function – functions like **collect.models()** and **mark.wrapper()** can be limited to that range of objects for selection. The function **collect.models()** is not particularly clever and it is possible for it to unknowingly aggregate analyses from different data sets if they have the same name. It does recognize when it is aggregating models of different type (e.g., **CJS** and **POPAN**) and will issue a warning. It will also issue a warning if the name of the processed data varies between models being collected but if the data are different but use the same object name it does not discriminate. However, if the models are collected within a function it will only collect those defined within the function preventing any unforeseen problems. Also, **mark.wrapper** works well within functions to define the set of models to run and we will demonstrate it here with the analysis of the swift data set from chapter 4.

end sidebar

We will use the swift dataset (Chapter 4) to demonstrate the use of scripts with functions. In the swift example, φ is thought to vary by **colony**, by **time**, or by **colony** and **time** (**colony*time**) because one **colony** has been classified as poor and the other as good. Capture probability p is thought to be either constant or vary by time. All of the pairings are considered for a total of 6 models to evaluate. Such a scheme is exactly how **mark.wrapper** was designed to operate. A set of specifications is given for each parameter and all possible combinations of the specifications of the parameters in the model (e.g., φ and p for CJS) are created for analysis. A function **create.model.list** identifies the model specifications by collecting any object named with a parameter name from the particular model followed by a period and any text description can follow the period. This is why we chose to name parameter specifications like **Phi.time** and **p.dot** as we did. Because it will collect any such objects it is best to use **create.model.list** within a function such that it will only collect those defined within the function.

Below we define the script that we created to analyze the swift data. The script imports the data, creates and runs the models, adjusts for over-dispersion and model averages the parameters. We have used comments identified by text following a # sign to document our analysis. We recommend liberal use of comments to help you understand what you were doing and thinking at the time that you created an analysis.

```
# Swift.R
#
# CJS analysis of the swift data from Chapter 4 of Cooch and White
#
# Import data (aa.inp) and convert it from the MARK inp file format to the \textbf{RMark}
# format using the function convert.inp  It is defined with 2 groups:
# Poor and Good to describe the quality of the colony.  This structure is defined
# with the group.df argument of convert.inp.  It expects that the file aa.inp is
```

```
# in the same directory as the current workspace.
#
aa=convert.inp("aa",group.df=data.frame(colony=c("Poor","Good")))
#
# Next create the processed dataframe and the design data.  We'll use a group
# variable for colony so it can be used in the set of models for Phi.  Factor
# variables (covariates with a small finite set of values) are best handled by using
# them to define groups in the data.
#
  aa.process=process.data(aa,model="CJS",groups="colony")
  aa.ddl=make.design.data(aa.process)
#
# Next create the function that defines and runs the set of models and returns
# a marklist with the results and a model.table.  It does not have any arguments
# but does use the aa.process and aa.ddl objects created above in the workspace
# The function create.model.list is the function that creates a dataframe of the
# names of the parameter specifications for each parameter in that type of model.
# If none are given for any parameter, the default specification will be used for
# that parameter in mark.  We used the adjust=FALSE argument because we know that
# the models time in Phi and p have extra parameters so we will accept the parameter
# counts from MARK and not adjust them.  The first argument of mark.wrapper is the
# model list of parameter specifications.  Remaining arguments that are passed to
# mark must be specified using the argument=value specification because the arguments
# of mark were not repeated in mark.wrapper so they must be passed using the
# argument=value syntax.
#
aa.models=function()
{
  Phi.colony=list(formula=~colony)
  Phi.time=list(formula=~time)
  Phi.colony.time=list(formula=~time*colony)
  p.dot=list(formula=~1)
  p.time=list(formula=~time)
  cml=create.model.list("CJS")
  results=mark.wrapper(cml,data=aa.process,ddl=aa.ddl,adjust=FALSE)
  return(results)
}
#
# Next run the function to create the models and store the results in
# aa.results which is a marklist.
#
aa.results=aa.models()
#
# Adjust for estimated overdispersion of chat=1.127
#
aa.results=adjust.chat(1.127,aa.results)
#
# Compute model averaged parameters
#
aa.model.avg.p=model.average(aa.results,"p",vcv=TRUE)
aa.model.avg.Phi=model.average(aa.results,"Phi",vcv=TRUE)
```

The table of model results is the same as that shown in chapter 4.

```
> aa.results
                        model npar    QAICc DeltaQAICc       weight QDeviance  chat
2        Phi(~colony)p(~time)    9 330.2689   0.000000 7.058575e-01  99.08106 1.127
1           Phi(~colony)p(~1)    3 332.1239   1.855043 2.791898e-01 113.75240 1.127
5             Phi(~time)p(~1)    8 338.1891   7.920167 1.345472e-02 109.19262 1.127
6          Phi(~time)p(~time)   13 342.8825  12.613639 1.287360e-03 102.69603 1.127
3    Phi(~time * colony)p(~1)   15 346.6140  16.345062 1.992654e-04 101.78298 1.127
4 Phi(~time * colony)p(~time)   20 352.3334  22.064458 1.141513e-05  95.44214 1.127
```

As well the model averaged capture probabilities shown in Chapter 4 are given below and the results are accurate to 7 places to the right of the decimal. Note that **RMark** only provides the unconditional standard error and the confidence interval based on it and does not provide the percentage due to model uncertainty.

```
> aa.model.avg.p$estimates[1:2,]
  par.index  estimate         se       lcl       ucl fixed group cohort age time Cohort Age Time colony
1        57 0.7502636 0.12064459 0.4698781 0.9732465        Good      1   1    2      0   1    0   Good
2        58 0.7230819 0.09321616 0.5118358 0.8667183        Good      1   2    3      0   2    1   Good
```

begin sidebar

Automating annual survey analyses

The swift example with the models we defined above is not a particularly involved analysis and it does not require much additional work with the standard **MARK** interface because each of the models we used are within the pre-defined set of models. However, even in this case, the **RMark** interface does have an advantage if the data set is routinely updated with an additional year of data. If you add a year of data with the standard **MARK** interface, you have to start from scratch to re-create the **MARK** project and the defined set of models; whereas, with the **RMark** interface it would only require re-running the script after changing the data. In some cases, it may be necessary to modify the script but in most cases even that will not be necessary because the PIM and design data structure are recreated automatically with the new data structure that adds another occasion. **R** has numerous ways of importing data including packages that provide direct access into EXCEL and ACCESS databases. This enables creating a script that requires no user intervention after the data are updated in the appropriate database. The ability to run **R** in batch mode with scripts opens the door to developing an interactive user interface that would run **RMark** with **R** and automate the script development. Such a system is currently being used with an **R** package for distance sampling analysis.

end sidebar

C.14. Defining groups with more than one variable

So far the examples we have shown did not really expand on the pre-defined models in the **MARK** interface except for the use of age and cohort. The pre-defined models in **MARK** include group (**g**) as one of the factors but what happens when groups are composed of two or more factor variables? Consider the **multi_group.inp** example described in Chapter 6 which has 6 sampling occasions and groups defined by **colony** and **sex**. If you include **g** in a model for these data, it will fit 4 parameters for **Poor-Female**, **Good-Female**, **Poor-Male**, and **Good-Male**. Fitting **g** is equivalent to fitting **~colony*sex** which is the full interaction model for **colony** and **sex**. Within the pre-defined models in **MARK** there is no capacity to fit any of the sub-models: **~colony**, **~sex**, and **~colony+sex** and to fit those models you need to create a design matrix which is described in chapter 6. When you jump to **g*t** models, fitting sub-models becomes even more important. What if capture probability varied by time and colony and survival varied by sex and time? Both of these are sub-models of the **g*t** pre-defined model and require a design matrix.

This is where the formula notation and automatic design matrix development starts to become quite useful. Once the group variables are defined, creating the full interaction model and the sub-models requires no more work than any of the other models that we have developed so far.

Below is a script that provides an analysis with a variety of models:

```
# Multi_group.R
#
# CJS analysis of the multi_group data from Chapter 6 of Cooch and White
#
# Import data (multi_group.inp) and convert it from the MARK inp file format to the RMark
# format using the function convert.inp  It is defined with 4 groups:
# Poor-Female, Good-Female, Poor-Male and Good-Male to describe the q
# quality of the colony and 2 sexes.  This structure is defined
# with the group.df argument of convert.inp which has 4 rows and 2 fields sex and colony
#
multigroup=convert.inp("multi_group",
 group.df=data.frame(sex=c(rep("Female",2),rep("Male",2)),colony=rep(c("Good","Poor"),2)))

#
# Next create the processed dataframe and the design data.  We'll use a group
# variable for colony so it can be used in the set of models for Phi.  Factor
# variables (covariates with a small finite set of values) are best handled by using
# them to define groups in the data.
#
multigroup.process=process.data(multigroup,model="CJS",groups=c("sex","colony"))
multigroup.ddl=make.design.data(multigroup.process)
#
# Next create the function that defines and runs the set of models and returns
# a marklist with the results and a model.table.
#
multigroup.models=function()
{
  Phi.colony=list(formula=~colony)
  Phi.sex=list(formula=~sex)
  Phi.sex.plus.colony=list(formula=~sex+colony)
  Phi.sex.time.plus.colony=list(formula=~sex*time+colony)
  p.time=list(formula=~time)
  p.colony.plus.sex=list(formula=~colony+sex)
  p.colony.time=list(formula=~time*colony)
  cml=create.model.list("CJS")
  results=mark.wrapper(cml,data=multigroup.process,ddl=multigroup.ddl)
  return(results)
}
#
# Next run the function to create the models and store the results in
# multigroup.results which is a marklist.
#
multigroup.results=multigroup.models()
#
# Compute model averaged parameters
#
multigroup.model.avg.p=model.average(multigroup.results,"p")
multigroup.model.avg.Phi=model.average(multigroup.results,"Phi")
```

The results table from the run is:

```
> multigroup.results
                                        model npar     AICc  DeltaAICc     weight  Deviance
12          Phi(~sex * time + colony)p(~time)   13 15440.75   0.000000 0.95888873 100.41075
11 Phi(~sex * time + colony)p(~time * colony)   17 15447.95   7.198505 0.02622000  99.58174
10  Phi(~sex * time + colony)p(~colony + sex)   12 15449.08   8.330000 0.01489127 110.74725
9                  Phi(~sex + colony)p(~time)    7 15907.94 467.182000 0.00000000 579.62086
8         Phi(~sex + colony)p(~time * colony)   11 15912.68 471.927000 0.00000000 576.34862
6                          Phi(~sex)p(~time)    6 15936.95 496.191000 0.00000000 610.63318
5                  Phi(~sex)p(~time * colony)   10 15938.44 497.686000 0.00000000 604.11265
3                       Phi(~colony)p(~time)    6 15996.61 555.860000 0.00000000 670.30226
2               Phi(~colony)p(~time * colony)   10 16000.36 559.606000 0.00000000 666.03275
7          Phi(~sex + colony)p(~colony + sex)    6 16004.56 563.801000 0.00000000 678.24333
4                   Phi(~sex)p(~colony + sex)    5 16031.57 590.818000 0.00000000 707.26243
1                Phi(~colony)p(~colony + sex)    5 16079.25 638.493000 0.00000000 754.93785
```

There is quite a jump in ΔAIC_c (**DeltaAICc**) from model 10 to model 9. This could be from exclusion of ***time** in φ_i or may be due to lack of convergence. Because model 9 is simpler than models 10-12, the latter is unlikely but we will use this as an opportunity to show how you can easily re-run a model using initial values from another model. The following uses the function **rerun.mark** to re-run model 9 using initial values from model 12 and stores the result back into the **marklist** in position 9:

```
> multigroup.results[[9]]=rerun.mark(multigroup.results[[9]],
   data=multigroup.process,ddl=multigroup.ddl,initial=multigroup.results[[12]])
```

A quick look at the summary output reveals the identical AIC_c values so the model converged to the same values. If the value had changed, we would have had to reconstruct the **model.table** as shown later. When we use **model.average** to obtain model-averaged real parameters we get a warning message that model 11 was dropped because some of the beta variances were negative:

```
> multigroup.model.avg.p=model.average(multigroup.results,"p")
Model  11 dropped from the model averaging because one or more beta variances
   are not positive
> multigroup.model.avg.Phi=model.average(multigroup.results,"Phi")
Model  11 dropped from the model averaging because one or more beta variances
   are not positive
```

Negative variances for the β's are symptomatic of something amiss so those models are dropped by default primarily as a way to draw attention to the issue. Negative variances are set to zero in **MARK** so they show up with an SE=0.00000 in the output and this behavior is mimicked in **RMark**, In this case, the negative variances occur because one of the parameters is at a boundary. φ for females at time 4 is 1 and this probably occurs because of confounding between φ and p for time 4. **MARK** reported 16 of the 17 parameters were estimable and that beta 5 (female time 4) was singular. We can either re-run this model and set **adjust=FALSE** or we can use the function **adjust.parameter.count** to reset the count to 16 as follows:

```
> multigroup.results[[11]]=adjust.parameter.count(multigroup.results[[11]],16)

  Number of parameters adjusted from  17  to  16
  Adjusted AICc=15445.94
  Unadjusted AICc =  15445.95
```

Once the number of parameters has been adjusted the model of table results must be recalculated:

```
  multigroup.results$model.table=model.table(multigroup.results)
 multigroup.results
                                        model npar     AICc  DeltaAICc     weight  Deviance
12          Phi(~sex * time + colony)p(~time)   13 15440.75   0.000000 0.91731475 100.41075
11 Phi(~sex * time + colony)p(~time * colony)   16 15445.94   5.190998 0.06843961  99.58174
10  Phi(~sex * time + colony)p(~colony + sex)   12 15449.08   8.330000 0.01424564 110.74725
9                  Phi(~sex + colony)p(~time)    7 15907.94 467.182000 0.00000000 579.62086
8         Phi(~sex + colony)p(~time * colony)   11 15912.68 471.927000 0.00000000 576.34862
6                           Phi(~sex)p(~time)    6 15936.95 496.191000 0.00000000 610.63318
5                  Phi(~sex)p(~time * colony)   10 15938.44 497.686000 0.00000000 604.11265
3                       Phi(~colony)p(~time)    6 15996.61 555.860000 0.00000000 670.30226
2               Phi(~colony)p(~time * colony)   10 16000.36 559.606000 0.00000000 666.03275
7          Phi(~sex + colony)p(~colony + sex)    6 16004.56 563.801000 0.00000000 678.24333
4                   Phi(~sex)p(~colony + sex)    5 16031.57 590.818000 0.00000000 707.26243
1                Phi(~colony)p(~colony + sex)    5 16079.25 638.493000 0.00000000 754.93785
```

The parameters can be model averaged across all models by using **drop=FALSE** as follows:

```
 > multigroup.model.avg.p=model.average(multigroup.results,"p",drop=FALSE)
Warning message:

Improper V-C matrix for beta estimates. Some variances non-positive.
 in: get.real(model.list[[i]], parameter, design = model.list[[i]]$design.matrix,

 > multigroup.model.avg.Phi=model.average(multigroup.results,"Phi",drop=FALSE)
Warning message:

Improper V-C matrix for beta estimates. Some variances non-positive.
 in: get.real(model.list[[i]], parameter, design = model.list[[i]]$design.matrix,
```

A warning message is given about the negative variances and clearly it does not make sense to consider model averaged estimates of φ for time 4 and p for time 5 but the remaining real parameters are unaffected.

C.15. More complex examples

Now we will consider some more complex examples that require more knowledge about designing formulas in situations where the factors are not fully crossed which means that some interactions do not exist in the data structure. We will use the example from Chapter 7 that uses **age.inp**. These data were derived from a study in which only young were marked and released (CJS design) but the young were then recaptured through time as they aged. With such a design not all ages are represented in all years so these factors are not fully crossed. A fully crossed design would have data for each combination of factors. In year 1 of the experiment there are only young birds that were just banded. In year 2 there are birds that are ages 0 and 1, in year 3 there are birds of ages 0 to 2, etc. The general solution to this type of problem is to create dummy variables (numeric 0/1 coding) and create interactions of effects using the **:** operator which includes the interactions without the main effects. This a very useful tool because it allows you to limit the range of an effect to the subset of parameters that have a value of 1 for the dummy variable. To understand this fully, look at the PIM chart in section 7.1.1 which shows an age by time model in which age is limited to 2 classes of young (age 0) and adult (age 1+). The structure

shows time varying probabilities for young with indices 1 to 6 and time varying probabilities for adults with indices 7 to 11. There are no adults at time 1 so there are only 5 survival probabilities for adults and 6 for young.

The PIM chart in 8.1.1 can be created with a formula by creating a 0/1 dummy variable for each age grouping. For example, let's assume that we created a dummy variable called **young** that is 1 for a **young** animal and 0 for an adult and then another variable called **adult** that is 1 for adult and 0 for young. If you construct a formula with the interaction of **young** and **time** (a factor variable), it will create a parameter for each time for **young** animals (indices 1 to 6) and by default it creates an intercept. To demonstrate this we will convert the input file, process the data and create the design data we need:

```
> #
> # Import data from age.inp file with convert.inp
> #
>   age=convert.inp("age")
> #Process data for CJS model
>   age.process=process.data(age,model="CJS")
> #Make default design data
>   age.ddl=make.design.data(age.process)
> #
> # Add a young/adult age field to the design data for Phi which we have named ya.
> # It uses right=FALSE so that the intervals are 0 (young) and 1 to 7 (adult).
> #
>   age.ddl=add.design.data(age.process,age.ddl,"Phi","age",bins=c(0,1,7),right=FALSE,name="ya")
> #
> # Add a field to the Phi design data that is equivalent except that it is a numeric
> # dummy coding variable with value 1 for young and 0 for adult; field is named young
> #
>   age.ddl$Phi$young=0
>   age.ddl$Phi$young[age.ddl$Phi$age==0]=1
> #
> # Likewise add an adult 0/1 numeric field to the Phi design data
> # which is simply =1-young
>   age.ddl$Phi$adult=1-age.ddl$Phi$young
```

Notice that we were able to create the **adult** field from the **young** field by subtraction. Now, let's show what **model.matrix** does with the formula ~**young:time** to give you a more complete understanding. First we will look at the non-simplified PIMS for φ by wrapping the default **mark** model call within a call to **PIMS**:

```
 > PIMS(mark(age,output=F),"Phi",simplified=F)
group = Group 1
   1  2  3  4  5  6
1  1  2  3  4  5  6
2     7  8  9 10 11
3       12 13 14 15
4          16 17 18
5             19 20
6                21
```

That shows there are 21 possible different parameters for φ which will match the number of rows in the following design matrix created by **`model.matrix`**:

```
> model.matrix(~young:time,age.ddl$Phi)
   (Intercept) young:time1 young:time2 young:time3 young:time4 young:time5 young:time6
1            1           1           0           0           0           0           0
2            1           0           0           0           0           0           0
3            1           0           0           0           0           0           0
4            1           0           0           0           0           0           0
5            1           0           0           0           0           0           0
6            1           0           0           0           0           0           0
7            1           0           1           0           0           0           0
8            1           0           0           0           0           0           0
9            1           0           0           0           0           0           0
10           1           0           0           0           0           0           0
11           1           0           0           0           0           0           0
12           1           0           0           1           0           0           0
13           1           0           0           0           0           0           0
14           1           0           0           0           0           0           0
15           1           0           0           0           0           0           0
16           1           0           0           0           1           0           0
17           1           0           0           0           0           0           0
18           1           0           0           0           0           0           0
19           1           0           0           0           0           1           0
20           1           0           0           0           0           0           0
21           1           0           0           0           0           0           1
 attr(,"assign")
[1] 0 1 1 1 1 1 1
 attr(,"contrasts")
 attr(,"contrasts")$time
[1] "contr.treatment"
```

The resulting design matrix is rather simple and with the exception of the intercept it is an identity matrix for indices (rows) 1,7,12,16,19,21 which are the φ parameters for young. The intercept is the φ parameter (β in link space) for adults and the time dependent probabilities for the young are the intercept plus the appropriate column for each time. So let's do the same with the **`adult`** field:

```
> model.matrix(~adult:time,age.ddl$Phi)
   (Intercept) adult:time1 adult:time2 adult:time3 adult:time4 adult:time5 adult:time6
1            1           0           0           0           0           0           0
2            1           0           1           0           0           0           0
3            1           0           0           1           0           0           0
4            1           0           0           0           1           0           0
5            1           0           0           0           0           1           0
6            1           0           0           0           0           0           1
7            1           0           0           0           0           0           0
8            1           0           0           1           0           0           0
9            1           0           0           0           1           0           0
10           1           0           0           0           0           1           0
11           1           0           0           0           0           0           1
12           1           0           0           0           0           0           0
13           1           0           0           0           1           0           0
14           1           0           0           0           0           1           0
15           1           0           0           0           0           0           1
```

```
16          1          0          0          0          0          0          0
17          1          0          0          0          0          1          0
18          1          0          0          0          0          0          1
19          1          0          0          0          0          0          0
20          1          0          0          0          0          0          1
21          1          0          0          0          0          0          0
 attr(,"assign")
[1] 0 1 1 1 1 1 1
 attr(,"contrasts")
 attr(,"contrasts")$time
[1] "contr.treatment"
```

Notice that the second column in the matrix is all zeros because there are no design data for an adult at time 1. The code in **RMark** simply removes any column containing all zeroes because it is not needed. For the matrix above, that would create 6 parameters for a model that had one constant young survival and 5 time dependent probabilities for adults; whereas ~**young:time** had 7 parameters with a constant adult survival and 6 time dependent probabilities for young.

So what happens if we use ~**young:time + adult:time** to try and construct the equivalent to the PIM chart in 8.1.1? To save space we won't show the entire design matrix but will show the following summaries:

```
 > dim(model.matrix(~young:time + adult:time,age.ddl$Phi))
[1] 21 13
 > colSums(model.matrix(~young:time + adult:time,age.ddl$Phi))
(Intercept) young:time1 young:time2 young:time3 young:time4 young:time5 young:time6
21           1           1           1           1           1           1
time1:adult time2:adult time3:adult time4:adult time5:adult time6:adult
 0           1           2           3           4           5
```

After deleting the one column of all zeroes, the resulting design matrix will still have 12 columns but there are only 11 unique parameters as shown in the 8.1.1 PIM chart. The solution is to remove the intercept which can be done by adding -1 to the formula. Below we show the same summaries using the correct formula:

```
 > dim(model.matrix(~-1+ young:time + adult:time,age.ddl$Phi))
[1] 21 12
 > colSums(model.matrix(~-1+young:time + adult:time,age.ddl$Phi))
young:time1 young:time2 young:time3 young:time4 young:time5 young:time6 time1:adult
1           1           1           1           1           1           0
time2:adult time3:adult time4:adult time5:adult time6:adult
1           2           3           4           5
```

After deleting the zero-sum column it will have the appropriate 11 parameters for the design matrix. Let's fit this model and examine the simplified PIM structure and design matrix.

```
  > Phi.yaxtime=list(formula=~-1+young:time+adult:time)
  > p.dot=list(formula=~1)
  > age.Phi.yaxtime.p.dot=mark(age.process,age.ddl,model.parameters=list(Phi=Phi.yaxtime,
        p=p.dot),output=FALSE)

 > PIMS(age.Phi.yaxtime.p.dot,"Phi")
group = Group 1
```

```
   1  2  3  4  5  6
1  1  2  3  4  5  6
2     7  3  4  5  6
3        8  4  5  6
4           9  5  6
5             10  6
6                11

 >  age.Phi.yaxtime.p.dot$design.matrix[,1:11]
    Phi:young:time1 Phi:young:time2 Phi:young:time3 Phi:young:time4 Phi:young:time5 Phi:young:time6
Phi g1 c1 a0 t1 "1"             "0"             "0"             "0"             "0"             "0"
Phi g1 c1 a1 t2 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c1 a2 t3 "0"             "0"             "0"             "0" n            "0"             "0"
Phi g1 c1 a3 t4 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c1 a4 t5 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c1 a5 t6 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c2 a0 t2 "0"             "1"             "0"             "0"             "0"             "0"
Phi g1 c3 a0 t3 "0"             "0"             "1"             "0"             "0"             "0"
Phi g1 c4 a0 t4 "0"             "0"             "0"             "1"             "0"             "0"
Phi g1 c5 a0 t5 "0"             "0"             "0"             "0"             "1"             "0"
Phi g1 c6 a0 t6 "0"             "0"             "0"             "0"             "0"             "1"
p g1 c1 a1 t2   "0"             "0"             "0"             "0"             "0"             "0"
       Phi:time2:adult Phi:time3:adult Phi:time4:adult Phi:time5:adult Phi:time6:adult p:(Intercept)
Phi g1 c1 a0 t1 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c1 a1 t2 "1"             "0"             "0"             "0"             "0"             "0"
Phi g1 c1 a2 t3 "0"             "1"             "0"             "0"             "0"             "0"
Phi g1 c1 a3 t4 "0"             "0"             "1"             "0"             "0"             "0"
Phi g1 c1 a4 t5 "0"             "0"             "0"             "1"             "0"             "0"
Phi g1 c1 a5 t6 "0"             "0"             "0"             "0"             "1"             "0"
Phi g1 c2 a0 t2 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c3 a0 t3 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c4 a0 t4 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c5 a0 t5 "0"             "0"             "0"             "0"             "0"             "0"
Phi g1 c6 a0 t6 "0"             "0"             "0"             "0"             "0"             "0"
p g1 c1 a1 t2   "0"             "0"             "0"             "0"             "0"             "1"
```

The numbering of the PIM is different but the structure is identical to the PIM chart in 8.1.1, Also, by rearranging the rows of the design matrix you could make it into an identity matrix because each row and each column have only a single 1.

Below is the script that we wrote to do the analysis above and fit other models for comparison. It includes models other than those fitted in Chapter 7.

```
#  markyoung_age.R  - script for fitting models for age.inp in which only young are marked
#
#
# Import data from age.inp file with convert.inp
#
  age=convert.inp("age")
#Process data for CJS model
  age.process=process.data(age,model="CJS")
#Make default design data
  age.ddl=make.design.data(age.process)
#
# Add a young/adult age field to the design data for Phi which we have named ya.
# It uses right=FALSE so that the intervals are 0 (young) and 1 to 7 (adult).
#
  age.ddl=add.design.data(age.process,age.ddl,"Phi","age",bins=c(0,1,7),right=FALSE,name="ya")
#
```

```
# Add a field to the Phi design data that is equivalent except that it is a numeric
# dummy coding variable with value 1 for young and 0 for adult; field is named young
#
  age.ddl$Phi$young=0
  age.ddl$Phi$young[age.ddl$Phi$age==0]=1
#
# Likewise add an adult 0/1 numeric field to the Phi design data which is simply =1-young
#
  age.ddl$Phi$adult=1-age.ddl$Phi$young

markyoung_age.models=function()
{
#Create formulas for Phi
# A constant survival model
  Phi.dot=list(formula=~1)
# A fully age depdendent but time invariant survival model
  Phi.age=list(formula=~age)
# A limited age model (young/adult) but time invariant survival model
  Phi.ya=list(formula=~ya)
# Limited age-time interaction survival model; young vary by time but adult
# survival is time invariant.  The intercept is the adult value
  Phi.yxtime.a=list(formula=~young:time)
# Fully age (young/adult) and time varying survival model with the time effect
# interacting with age. We cannot use ya*time because there are no adults for time1
# The -1 removes the intercept which is not needed because the young:time creates a
# parameter for each time for the young animals and the adult:time creates a parameter
# for each time that has adults.  It is equivalent to a PIM coding for the problem
# but still uses a design matrix.
  Phi.yaxtime=list(formula=~-1+young:time+adult:time)
#Create formulas for p
  p.dot=list(formula=~1)
  p.time=list(formula=~time)
#Create model list
  cml=create.model.list("CJS")
#Run and return complete set of models
  return(mark.wrapper(cml,data=age.process,ddl=age.ddl))
}
# Run analysis function and store in marklist
 > markyoung_age.results=markyoung_age.models()
```

Below is the model results table:

```
> markyoung_age.results
                                        model npar     AICc  DeltaAICc     weight Deviance
9                       Phi(~young:time)p(~1)    8 5050.502   0.000000 0.57098900 139.7729
7     Phi(~-1 + young:time + adult:time)p(~1)   12 5051.756   1.254100 0.30500249 132.9714
10                   Phi(~young:time)p(~time)   13 5053.875   3.372900 0.10573331 133.0730
8  Phi(~-1 + young:time + adult:time)p(~time)   17 5057.385   6.883649 0.01827521 128.5015
5                               Phi(~ya)p(~1)    3 5169.603 119.101100 0.00000000 268.9136
1                              Phi(~age)p(~1)    7 5170.234 119.731864 0.00000000 261.5153
2                           Phi(~age)p(~time)   12 5174.617 124.114840 0.00000000 255.8321
6                            Phi(~ya)p(~time)    8 5175.432 124.930200 0.00000000 264.7031
4                             Phi(~1)p(~time)    7 5385.402 334.900600 0.00000000 476.6840
3                                Phi(~1)p(~1)    2 5418.139 367.637300 0.00000000 519.4538
```

```
> #get summary of model 8 to see how it denotes parameter counts and AICc
> summary(markyoung_age.results[[8]])
Output summary for CJS model
Name : Phi(~-1 + young:time + adult:time)p(~time)

Npar :  17  (unadjusted=16)
-2lnL:  5023.183
AICc :  5057.385  (unadjusted=5055.3628)

Beta
                   estimate        se        lcl        ucl
Phi:young:time1 -1.2946578 0.1785905 -1.6446952 -0.9446205
Phi:young:time2  0.2484213 0.1388986 -0.0238200  0.5206626
Phi:young:time3  0.1464425 0.1308574 -0.1100380  0.4029230
Phi:young:time4 -0.8908855 0.1337821 -1.1530983 -0.6286726
Phi:young:time5 -0.8497168 0.1327969 -1.1099988 -0.5894348
Phi:young:time6 -0.9103939 0.0000000 -0.9103939 -0.9103939
Phi:time2:adult  0.2003894 0.3472809 -0.4802813  0.8810600
Phi:time3:adult  1.1011872 0.2531575  0.6049984  1.5973760
Phi:time4:adult  1.0734292 0.2089632  0.6638612  1.4829971
Phi:time5:adult  0.8498674 0.1874681  0.4824300  1.2173048
Phi:time6:adult  1.2672109 0.0000000  1.2672109  1.2672109
p:(Intercept)    0.4519732 0.3418917 -0.2181345  1.1220810
p:time3          0.0452004 0.3796693 -0.6989513  0.7893522
p:time4          0.1410901 0.3679322 -0.5800569  0.8622371
p:time5          0.1825269 0.3691351 -0.5409778  0.9060316
p:time6          0.4714351 0.3766168 -0.2667339  1.2096041
p:time7          0.3346337 0.0000000  0.3346337  0.3346337


Real Parameter Phi

          1         2         3         4         5         6
1 0.2150655 0.5499304 0.7504825 0.7452485 0.7005393 0.7802649
2           0.5617879 0.7504825 0.7452485 0.7005393 0.7802649
3                     0.5365453 0.7452485 0.7005393 0.7802649
4                               0.2909271 0.7005393 0.7802649
5                                         0.2994923 0.7802649
6                                                   0.2869192


Real Parameter p

          2         3         4         5         6         7
1 0.6111083 0.6217949 0.6440677 0.6535092 0.7157361 0.6871023
2           0.6217949 0.6440677 0.6535092 0.7157361 0.6871023
3                     0.6440677 0.6535092 0.7157361 0.6871023
4                               0.6535092 0.7157361 0.6871023
5                                         0.7157361 0.6871023
6                                                   0.6871023

> # show model.table using parameter counts from MARK
> model.table(markyoung_age.results[1:10],adjust=F)
                                        model npar     AICc DeltaAICc     weight Deviance
```

```
9                        Phi(~young:time)p(~1)    8 5050.502    0.0000 0.55330256 139.7729
7     Phi(~-1 + young:time + adult:time)p(~1)   12 5051.756    1.2541 0.29555501 132.9714
10                   Phi(~young:time)p(~time)   13 5053.875    3.3729 0.10245821 133.0730
8  Phi(~-1 + young:time + adult:time)p(~time)   16 5055.363    4.8611 0.04868422 128.5015
1                               Phi(~age)p(~1)    6 5168.224  117.7226 0.00000000 261.5153
5                                Phi(~ya)p(~1)    3 5169.603  119.1011 0.00000000 268.9136
2                           Phi(~age)p(~time)   11 5172.601  122.0989 0.00000000 255.8321
6                            Phi(~ya)p(~time)    8 5175.432  124.9302 0.00000000 264.7031
4                             Phi(~1)p(~time)    7 5385.402  334.9006 0.00000000 476.6840
3                                 Phi(~1)p(~1)    2 5418.139  367.6373 0.00000000 519.4538
```

This final results table has the same values as the equivalent table in Chapter 7 except that it contains more models including the best models.

Now let's take the next step presented in chapter 7 and consider the situation in which both young and adults are marked and released. The primary goal of this exercise is to evaluate whether adult survival differs for the 2 groups: marked as young versus marked as adult.

```
age_ya=convert.inp("age_ya",group.df=data.frame(age=c("Young","Adult")))
# Process data for CJS model; an initial age is defined as 1 for adults and 0
# for young.  They are assigned in that order because they are assigned in order of
# the factor variable which is alphabetical with adult before young.  It does not
# matter that adults could be a mixture of ages because we will only model young (0)
# and adult (1+).
age_ya.process=process.data(age_ya,group="age",initial.age=c(1,0))
# Make the default design data
age_ya.ddl=make.design.data(age_ya.process)
#
# Add a young/adult age field to the design data for Phi which we have named ya.
# It uses right=FALSE so that the intervals are 0 (young) and 1 to 7 (adult).
#
age_ya.ddl=add.design.data(age_ya.process,age_ya.ddl,"Phi","age",bins=c(0,1,7),
          right=FALSE,name="ya")
#
# Next create a dummy field called marked.as.adult which is 0 for the group
# marked as young and 1 for the group marked as adults.
#
age_ya.ddl$Phi$marked.as.adult=0
age_ya.ddl$Phi$marked.as.adult[age_ya.ddl$Phi$group=="Adult"]=1
```

Look through the design data for φ so you understand how each of the added fields are defined. Pay particular attention to the difference between the **ya** field and **marked.as.adult**. The field **ya** represents age classes and they change over time for an individual marked and released as young whereas the **marked.as.adult** is a dummy variable for the grouping and it is static.

```
> age_ya.ddl$Phi
  group cohort age time Cohort Age Time initial.age.class    ya marked.as.adult
1 Adult      1   1    1      0   1    0             Adult [1,7]               1
2 Adult      1   2    2      0   2    1             Adult [1,7]               1
3 Adult      1   3    3      0   3    2             Adult [1,7]               1
4 Adult      1   4    4      0   4    3             Adult [1,7]               1
5 Adult      1   5    5      0   5    4             Adult [1,7]               1
6 Adult      1   6    6      0   6    5             Adult [1,7]               1
7 Adult      2   1    2      1   1    1             Adult [1,7]               1
8 Adult      2   2    3      1   2    2             Adult [1,7]               1
```

```
9  Adult      2   3    4      1   3    3             Adult [1,7]               1
10 Adult      2   4    5      1   4    4             Adult [1,7]               1
11 Adult      2   5    6      1   5    5             Adult [1,7]               1
12 Adult      3   1    3      2   1    2             Adult [1,7]               1
13 Adult      3   2    4      2   2    3             Adult [1,7]               1
14 Adult      3   3    5      2   3    4             Adult [1,7]               1
15 Adult      3   4    6      2   4    5             Adult [1,7]               1
16 Adult      4   1    4      3   1    3             Adult [1,7]               1
17 Adult      4   2    5      3   2    4             Adult [1,7]               1
18 Adult      4   3    6      3   3    5             Adult [1,7]               1
19 Adult      5   1    5      4   1    4             Adult [1,7]               1
20 Adult      5   2    6      4   2    5             Adult [1,7]               1
21 Adult      6   1    6      5   1    5             Adult [1,7]               1
22 Young      1   0    1      0   0    0             Young [0,1)               0
23 Young      1   1    2      0   1    1             Young [1,7]               0
24 Young      1   2    3      0   2    2             Young [1,7]               0
25 Young      1   3    4      0   3    3             Young [1,7]               0
26 Young      1   4    5      0   4    4             Young [1,7]               0
27 Young      1   5    6      0   5    5             Young [1,7]               0
28 Young      2   0    2      1   0    1             Young [0,1)               0
29 Young      2   1    3      1   1    2             Young [1,7]               0
30 Young      2   2    4      1   2    3             Young [1,7]               0
31 Young      2   3    5      1   3    4             Young [1,7]               0
32 Young      2   4    6      1   4    5             Young [1,7]               0
33 Young      3   0    3      2   0    2             Young [0,1)               0
34 Young      3   1    4      2   1    3             Young [1,7]               0
35 Young      3   2    5      2   2    4             Young [1,7]               0
36 Young      3   3    6      2   3    5             Young [1,7]               0
37 Young      4   0    4      3   0    3             Young [0,1)               0
38 Young      4   1    5      3   1    4             Young [1,7]               0
39 Young      4   2    6      3   2    5             Young [1,7]               0
40 Young      5   0    5      4   0    4             Young [0,1)               0
41 Young      5   1    6      4   1    5             Young [1,7]               0
42 Young      6   0    6      5   0    5             Young [0,1)               0
>
```

Before we go too far with this example, we'll show the simplified PIMS for the ~**ya*time** model which we could not fit in the previous example but we can fit now because adults were marked at time 1.

```
 > PIMS(mark(age_ya.process,age_ya.ddl,model.parameters=list(Phi=list(formula=~ya*time)),
     output=F),"Phi")

group = ageAdult
   1  2  3  4  5  6
1  1  2  3  4  5  6
2     2  3  4  5  6
3        3  4  5  6
4           4  5  6
5              5  6
6                 6
group = ageYoung
   1  2  3  4  5  6
1  7  2  3  4  5  6
```

```
2      8  3  4  5  6
3         9  4  5  6
4           10  5  6
5              11  6
6                 12
```

The numbering is slightly different than what is shown in the second and final sets of PIMS from section 8.1.2, but if you look closely you'll see that the structure is identical with survival varying over time and interacting with age as defined by young/adult age classes. Hmm, that is quite close to what we want for the structure to evaluate whether adult survival is different between the 2 groups. All we really need to do is add **marked.as.adult** to the formula. Let's fit that model for φ and the sub-model given above and assume that capture probability varies by group but is time-invariant:

```
age_ya.models=function() {
Phi.ya.time.plus.marked.as.adult=list(formula=~ya*time+marked.as.adult)
Phi.ya.time=list(formula=~ya*time)
p.marked.as.adult=list(formula=~marked.as.adult)
cml=create.model.list("CJS")
results=mark.wrapper(cml,data=age_ya.process,ddl=age_ya.ddl,output=FALSE)
return(results) }

age_ya.results=age_ya.models()

 Variable  marked.as.adult  used in formula is not defined in data
 Error in make.mark.model(data.proc, title = title, covariates =
 covariates,  :

 Variable  marked.as.adult  used in formula is not defined in data
 Error in make.mark.model(data.proc, title = title, covariates =
 covariates,  :

 No mark models found Error in collect.models() :
```

What did we do wrong? We defined **marked.as.adult** and the spelling and punctuation is correct. You will make this mistake which is why we showed it. The error message could be made better because it does not tell you where the problem occurs, but remember that design data is specific to each parameter and we only defined the **marked.as.adult** field for φ but we just used it above for the formula for p. That is the problem. One solution would be to use **~group** for the formula for p because that will give the same model with a slightly different parameterization. Another solution is to create the design data as follows and re-run the analysis:

```
  > age_ya.ddl$p$marked.as.adult=0
  > age_ya.ddl$p$marked.as.adult[age_ya.ddl$p$group=="Adult"]=1
  > age_ya.results=age_ya.models()

  > age_ya.results
                                                 model npar     AICc DeltaAICc    weight Deviance
2 Phi(~ya * time + marked.as.adult)p(~marked.as.adult)   15 13846.48     0.000 0.5304622 274.5737
1                   Phi(~ya * time)p(~marked.as.adult)   14 13846.72     0.244 0.4695378 276.8261
```

Did we get the models and parameter counts correct? With the **~ya*time** model shown in the final set of PIMS in 8.1.2 there are 12 parameters for φ and for our model with p there are 2 parameters (one for each group) so that is 14 and it matches the count for model 1. Our model 2 adds a single parameter for φ so that makes 15 also matching the results. If we look at the simplified PIMS for φ with model 2 we see that the structure matches the PIMS laid out for this problem with 17 indices, but they are not numbered in the same order:

```
 > PIMS(age_ya.results[[2]],"Phi")
group = ageAdult
   1  2  3  4  5  6
1  1  2  3  4  5  6
2     2  3  4  5  6
3        3  4  5  6
4           4  5  6
5              5  6
6                 6
group = ageYoung
   1  2  3  4  5  6
1  7  8  9 10 11 12
2    13  9 10 11 12
3       14 10 11 12
4          15 11 12
5             16 12
6                17
```

The design matrix also does not match the one shown in 8.1.2 because the rows are ordered differently and the effects are parameterized differently so the betas will be different but the real parameters would be the same. The design matrix shown below is a cosmetically edited version of the contents contained in **age_ya.results[[2]]$design.matrix** to make it more visually apparent. The design matrix is stored as a matrix of strings so the "" were removed, the **marked.as.adult** column was moved over, the column headers were renamed to use adult rather than the factor **ya:[1,7]**. The intercept (the first column) is for young- time1 which is apparent when you see that row 7 (the index for this parameter) is the one with a single 1 in the row. The second column is the additive age-effect for adult survival and the third column is the **marked.as.adult** effect which is 1 for only the first 6 rows (indices 1-6). Columns 4-8 are baseline time effects for times 2 to 6. Finally, columns 9-13 are the interaction of time with age for adults. All of these columns would be the same for model 1 except that column 3 would not be included.

```
Adult   Adult   1   1   1   1   0   0   0   0   0   0   0   0   0   0
Adult   Adult   2   1   1   1   1   0   0   0   0   1   0   0   0   0
Adult   Adult   3   1   1   1   0   1   0   0   0   0   1   0   0   0
Adult   Adult   4   1   1   1   0   0   1   0   0   0   0   1   0   0
Adult   Adult   5   1   1   1   0   0   0   1   0   0   0   0   1   0
Adult   Adult   6   1   1   1   0   0   0   0   1   0   0   0   0   1
Young   Young   7   1   0   0   0   0   0   0   0   0   0   0   0   0
Young   Adult   8   1   1   0   1   0   0   0   0   1   0   0   0   0
Young   Adult   9   1   1   0   0   1   0   0   0   0   1   0   0   0
Young   Adult   10  1   1   0   0   0   1   0   0   0   0   1   0   0
Young   Adult   11  1   1   0   0   0   0   1   0   0   0   0   1   0
Young   Adult   12  1   1   0   0   0   0   0   1   0   0   0   0   1
Young   Young   13  1   0   0   1   0   0   0   0   0   0   0   0   0
```

```
Young    Young    14  1   0   0   0   1   0   0   0   0   0   0   0   0
Young    Young    15  1   0   0   0   0   1   0   0   0   0   0   0   0
Young    Young    16  1   0   0   0   0   0   1   0   0   0   0   0   0
Young    Young    17  1   0   0   0   0   0   0   1   0   0   0   0   0
```

It is also useful to distinguish here between TSM (time since marking) and age models. This distinction is made based on the initial age that is assigned to groups. If the initial ages for the groups are identical (and technically 0) then age in the design data is really TSM. Age and TSM are the same when everything is the same age at marking like in the example when only young were marked. If you assign different initial ages to groups to represent actual age, you can still define a TSM field in the design data as age-initial age but make sure to use numeric values like Age or convert factors to numeric values to do the calculation.

Let's go back to the dipper data to show some more complications that can arise when the design is not fully crossed. In this case, we will assume that dippers are all released at age 0 and we expect that survival is time dependent for young (age 0) but not for all adults (1+). Also, we expect age differences in adult survival and we expect that the age differences might be different for males and females. Also, we expect that adult capture probability changes when they reach age 2 for females and age 3 for males. This is most likely bogus for dippers but then again it is just an example. So how do we go about building a set of models? First, we need to set up the design data that we need for the structure that we have identified. The following code processes the data, makes the default design data and then creates fields adult (0/1) and young (0/1) in the ψ design data and the variable shift (0/1) in p which was defined to create a sex-specific timing of a shift in capture probability possibly associated with the onset of breeding age.

```
> dipper.processed=process.data(dipper,begin.time=1980,groups="sex")
> dipper.ddl=make.design.data(dipper.processed)
> dipper.ddl$Phi$adult=0
> dipper.ddl$Phi$adult[dipper.ddl$Phi$age>=1]=1
> dipper.ddl$Phi$adult[dipper.ddl$Phi$Age>=1]=1
> dipper.ddl$Phi$young=1- dipper.ddl$Phi$adult
> dipper.ddl$Phi
> dipper.ddl$p$shift=0
> dipper.ddl$p$shift[dipper.ddl$p$Age>=3&dipper.ddl$p$sex=="Male"]=1
> dipper.ddl$p$shift[dipper.ddl$p$Age>=2&dipper.ddl$p$sex=="Female"]=1
> dipper.ddl$p
```

With these fields defined we can consider how to construct formula for various models that we propose. First we will consider the φ models and we will use the **R** functions **`model.matrix`** and **`colSums`** to examine how the model is constructed. Using **`model.matrix`** within **`colSums`** will show the columns in the design matrix and if they are non-zero. For example, if we want time dependent survival for the young we could do as follows:

```
> colSums(model.matrix(~young:time,dipper.ddl$Phi))
(Intercept) young:time1980 young:time1981 young:time1982 young:time1983 young:time1984 young:time1985
         42              2              2              2              2              2              2
```

This formula would create 6 parameters for young survival and then an intercept which would apply to adults which would have a constant survival. If we wanted to add an age and sex dependent survival for adults it would look as follows:

```
> colSums(model.matrix(~young:time+adult:age:sex,dipper.ddl$Phi))
        (Intercept)       young:time1980       young:time1981       young:time1982       young:time1983
```

```
                  42                    2                    2                    2                    2
      young:time1984       young:time1985 adult:age0:sexFemale adult:age1:sexFemale adult:age2:sexFemale
                   2                    2                    0                    5                    4
adult:age3:sexFemale adult:age4:sexFemale adult:age5:sexFemale   adult:age0:sexMale   adult:age1:sexMale
                   3                    2                    1                    0                    5
  adult:age2:sexMale   adult:age3:sexMale   adult:age4:sexMale   adult:age5:sexMale
                   4                    3                    2                    1
```

However, it has 17 non-zero columns but we only need 16 parameters (6 for age 0 and 5 each for the ages 1-5 for both male and female). The solution as noted above was to use the -1 to remove the intercept to get 16 parameters:

```
> colSums(model.matrix(~-1+young:time+adult:age:sex,dipper.ddl$Phi))
      young:time1980       young:time1981       young:time1982       young:time1983       young:time1984
                   2                    2                    2                    2                    2
      young:time1985 adult:age0:sexFemale adult:age1:sexFemale adult:age2:sexFemale adult:age3:sexFemale
                   2                    0                    5                    4                    3
adult:age4:sexFemale adult:age5:sexFemale   adult:age0:sexMale   adult:age1:sexMale   adult:age2:sexMale
                   2                    1                    0                    5                    4
  adult:age3:sexMale   adult:age4:sexMale   adult:age5:sexMale
                   3                    2                    1
```

Now, what if we wanted a model with age effects and an additive sex effect solely for adults:

```
> colSums(model.matrix(~-1+young:time+adult:age+adult:sex,dipper.ddl$Phi))
young:time1980 young:time1981 young:time1982 young:time1983 young:time1984 young:time1985     adult:age0
             2              2              2              2              2              2              0
    adult:age1     adult:age2     adult:age3     adult:age4     adult:age5  adult:sexMale
            10              8              6              4              2             15
```

That works as expected with 12 non-zero columns for the 12 parameters (6 for young, 5 for ages and 1 additive sex effect (male) for the adult age classes.

However, if we wanted an additive sex effect for each age including young, things go awry:

```
> colSums(model.matrix(~-1+young:time+adult:age+sex,dipper.ddl$Phi))
     sexFemale        sexMale young:time1980 young:time1981 young:time1982 young:time1983 young:time1984
            21             21              2              2              2              2              2
young:time1985     adult:age0     adult:age1     adult:age2     adult:age3     adult:age4     adult:age5
             2              0             10              8              6              4              2
```

because the -1 does not remove the intercept and it simply changes the design matrix to have separate intercepts for each sex and we end up with 13 parameters instead of 12 as above. Although it will not affect **`model.matrix`**, the solution for **RMark** is to set the argument **`remove.intercept=TRUE`** in the parameter specification as shown below. That will force removal of the intercept and can always be used in place of the -1 in a formula. If you use **`remove.intercept=TRUE`**, do not use the -1 in the formula.

On the next page is the script for this analysis which examines a sequence of models for φ including those above and a sequence for p including the shift in p. Given that this example was contrived it should be surprising that these imaginary models were not particularly good ones, but we show the results to demonstrate that the number of parameters were correct.

```
do.complicated.dipper.models=function()
{
# retrieve data, process it for CJS model and make default design data
    data(dipper)
    dipper.processed=process.data(dipper,begin.time=1980,groups="sex")
    dipper.ddl=make.design.data(dipper.processed)
# create additional Phi fields adult and young
    dipper.ddl$Phi$adult=0
```

```
    dipper.ddl$Phi$adult[dipper.ddl$Phi$Age>=1]=1
    dipper.ddl$Phi$young=1- dipper.ddl$Phi$adult
# create additional p field for sex-specific shift in p at "breeding" age
    dipper.ddl$p$shift=0
    dipper.ddl$p$shift[dipper.ddl$p$Age>=3&dipper.ddl$p$sex=="Male"]=1
    dipper.ddl$p$shift[dipper.ddl$p$Age>=2&dipper.ddl$p$sex=="Female"]=1
# define models for Phi
    Phi.dot=list(formula=~1)
    Phi.ytime=list(formula=~young:time)
    Phi.ytime.plus.adultxagexsex=list(formula=~young:time+adult:age:sex,remove.intercept=TRUE)
    Phi.ytime.plus.adultxage.plussex=list(formula=~young:time+adult:age+sex,remove.intercept=TRUE)
    Phi.ytime.plus.adultxage.plusadultxsex=list(formula=~young:time+adult:age+adult:sex,
                                                remove.intercept=TRUE)
# define models for p
    p.dot=list(formula=~1)
    p.time=list(formula=~time)
    p.shift=list(formula=~shift)
    p.shiftxsex=list(formula=~shift*sex)
# create model list
    cml=create.model.list("CJS")
# run and return models
    return(mark.wrapper(cml,data=dipper.processed,ddl=dipper.ddl))
}

complicated.results=do.complicated.dipper.models()
```

```
complicated.results

                                                  model npar     AICc DeltaAICc       weight Deviance
1                                        Phi(~1)p(~1)    2 670.8660  0.000000 5.877454e-01 84.36055
2                                    Phi(~1)p(~shift)    3 672.8926  2.026520 2.133713e-01 84.35857
5                               Phi(~young:time)p(~1)    8 674.6677  3.801640 8.783621e-02 75.84524
3                              Phi(~1)p(~shift * sex)    5 675.9918  5.125757 4.530490e-02 83.37182
6                           Phi(~young:time)p(~shift)    9 676.7273  5.861220 3.136472e-02 75.81745
4                                     Phi(~1)p(~time)    7 678.7481  7.882080 1.141872e-02 82.00306
9      Phi(~young:time + adult:age + adult:sex)p(~1)   13 679.7517  8.885695 6.913294e-03 70.39112
7                     Phi(~young:time)p(~shift * sex)   11 680.1781  9.312101 5.585887e-03 75.06334
13           Phi(~young:time + adult:age + sex)p(~1)   13 681.2700 10.403965 3.235913e-03 71.90939
10 Phi(~young:time + adult:age + adult:sex)p(~shift)   14 681.7379 10.871858 2.560916e-03 70.23888
8                           Phi(~young:time)p(~time)   13 682.5149 11.648835 1.736508e-03 73.15426
14       Phi(~young:time + adult:age + sex)p(~shift)   14 683.3693 12.503268 1.132763e-03 71.87029
17             Phi(~young:time + adult:age:sex)p(~1)   17 684.4892 13.623220 6.470601e-04 66.51214
11 Phi(~young:time+adult:age+adult:sex)p(~shift*sex)   16 684.9887 14.122623 5.040812e-04 69.18147
18         Phi(~young:time + adult:age:sex)p(~shift)   18 686.5849 15.718830 2.269283e-04 66.42716
15 Phi(~young:time + adult:age + sex)p(~shift * sex)   16 687.0127 16.146713 1.832209e-04 71.20556
12  Phi(~young:time + adult:age + adult:sex)p(~time)   18 687.9284 17.062380 1.159152e-04 67.77071
16        Phi(~young:time + adult:age + sex)p(~time)   18 689.2101 18.344070 6.106960e-05 69.05240
19   Phi(~young:time + adult:age:sex)p(~shift * sex)   20 689.8623 18.996264 4.407607e-05 65.31111
20          Phi(~young:time + adult:age:sex)p(~time)   22 692.6151 21.749106 1.112835e-05 63.62686
```

C.16. Individual covariates

As promised, we will now divulge the fourth trick in **RMark** which was needed to encompass individual covariates. You do not need to know how this trick works to use **RMark** and we are only describing it here in case someone wanted to use it in another similar application. If you look through the help file for **`make.mark.model`** you will see that there are arguments for a parameter specification called **`component`** and **`component.name`**. These arguments were included before the fourth trick was discovered and included. Now they are no longer needed. Those arguments were used to create additional columns that were pasted onto the design matrix to include individual covariates. This was done because individual covariates are entered into the design matrix for **MARK** as a string which contains the name of the covariate rather than 0 or 1 or other numeric value. There is no direct way to use **`model.matrix`** to do add these covariate names - thus the trick.

When **RMark** encounters an individual covariate (a name not in the design data), it creates a dummy variable in the design data for that covariate. The covariate name is used for the dummy variable name and it is given the value 1 for each row in the design data. Then the entire formula with the individual covariate and the modified design data is passed to model.matrix to create the design matrix which is only partially complete. **RMark** then processes the design matrix further to add the covariate names for **MARK**. Any columns with names that contain any individual covariate are modified in the following way: 1) any 0 values are left as is, 2) any value of 1 is changed to a string with the name of the covariate, and 3) if the value is neither 1 or 0, then it uses the product construct used in **MARK** design matrices and the value is replaced with the string "**`product(value,covariate_name)`**". The final step enables the use of formula containing interactions of individual covariates and design data covariates.

There is actually one more step that **RMark** does to enable time-varying covariates. If you use an individual covariate name that does not exist in the data, then it will look for variables that have that name as the prefix and a sequence of suffixes that match the values of the time variable in the design data for that particular parameter. This means that the variable names have to be constructed in a fashion that is consistent with the value chosen for begin.time and which is consistent with the labeling of times which is different for interval parameters such as φ and occasion parameters like p. If **RMark** finds a set of covariates that are properly named, then it constructs the design matrix using the covariate names that are appropriate for each row in the design matrix based on the value of the time field for that specific parameter.

Well with that said there is not much more to say about individual covariates except to show some examples that demonstrate how they are used in formula and how covariate-specific real parameter estimates can be computed after the model is fitted. To do that, we will continue to abuse the dipper data and create some imaginary weight data which was the weight of the bird at the time of first capture. We will fit models in which weight affects survival for all times for both sexes (**`weight`**) and then with a sex effect and sex-weight interaction (**`sex*weight`**). We will also show how the affect of the covariate can be limited to the first survival post-capture (**`young:weight`**). The following is the script that examines these and other models. Comments are given to explain each φ model.

```
# retrieve data, create some imaginary weight data using a random normal
> data(dipper)
> dipper$weight=rnorm(294,10,2)
> do.dipper.covariate.example=function()
{
# process the data for CJS model and make default design data
    dipper.processed=process.data(dipper,begin.time=1980,groups="sex")
    dipper.ddl=make.design.data(dipper.processed)
# create additional Phi fields adult and young
```

```
    dipper.ddl$Phi$adult=0
    dipper.ddl$Phi$adult[dipper.ddl$Phi$Age>=1]=1
    dipper.ddl$Phi$young=1- dipper.ddl$Phi$adult
# define models for Phi
    Phi.dot=list(formula=~1)
# weight only for all survivals
    Phi.weight=list(formula=~weight)
# sex and sex-dependent slope for weight
    Phi.weight.x.sex=list(formula=~weight*sex)
# same intercept for male/female with a sex-dependent slope for weight
    Phi.weight.sex=list(formula=~weight:sex)
# effect of weight only for first time post-capture; if you exclude the
# adult term, then an adult would have the intercept survival which would
# be the value for weight=0
    Phi.weight.plus.sex=list(formula=~adult + young:weight)
# define models for p
    p.dot=list(formula=~1)
    p.time=list(formula=~time)
# create model list
    cml=create.model.list("CJS")
# run and return models
    return(mark.wrapper(cml,data=dipper.processed,ddl=dipper.ddl))
    }
> covariate.results=do.dipper.covariate.example()
```

The results really do not matter because the example and data are bogus, but it is useful to examine the resulting design matrix that was constructed for some of these models. You can look at the simplified design matrix easily as follows:

```
 > covariate.results[[3]]$design.matrix
                          Phi:(Intercept) Phi:weight p:(Intercept)
Phi gFemale c1980 a0 t1980 "1"             "weight"   "0"
p gFemale c1980 a1 t1981   "0"             "0"        "1"

 > covariate.results[[5]]$design.matrix
                          Phi:(Intercept) Phi:adult Phi:young:weight p:(Intercept)
Phi gFemale c1980 a0 t1980 "1"             "0"       "weight"         "0"
Phi gFemale c1980 a1 t1981 "1"             "1"       "0"              "0"
p gFemale c1980 a1 t1981   "0"             "0"       "0"              "1"

 > covariate.results[[7]]$design.matrix
                          Phi:(Intercept) Phi:weight:sexFemale Phi:weight:sexMale p:(Intercept)
Phi gFemale c1980 a0 t1980 "1"             "weight"             "0"                "0"
Phi gMale c1980 a0 t1980   "1"             "0"                  "weight"           "0"
p gFemale c1980 a1 t1981   "0"             "0"                  "0"                "1"

 > covariate.results[[9]]$design.matrix
                          Phi:(Intercept) Phi:weight Phi:sexMale Phi:weight:sexMale p:(Intercept)
Phi gFemale c1980 a0 t1980 "1"             "weight"   "0"         "0"                "0"
Phi gMale c1980 a0 t1980   "1"             "weight"   "1"         "weight"           "0"
p gFemale c1980 a1 t1981   "0"             "0"        "0"         "0"                "1"
```

These are the simplified design matrices which are used after the PIMS have been recoded from all-different to the unique values. The non-simplified design matrix would contain 42 rows for φ and 42 rows for p. Notice that the resulting number of rows in the simplified design matrix depends on the formulas used which determine the unique number of parameters required.

It is useful to examine the design matrix to make sure you get the model you think that you specified with the formula. Even though **RMark** creates the PIMS and design matrix for you, it does not mean that you can shut off your brain and stop thinking. As an example, it would be very easy to make a mistake and specify the one model as **~young:weight**. At first glance that might seem to do what you want to restrict the effect of weight to the first capture φ and it does do that but it also stupidly assigns adult survival to the intercept which is the value where **weight=0**. Even though **RMark** removes the drudgery of creating design matrices, it does not eliminate the possibility of making a mistake by incorrect specification of the model. Examining the design matrix and using **model.matrix** (if there are no individual covariates) is the best way to prevent those mistakes.

Once you have fitted models using individual covariates, you will often want to compute predicted values at one or more covariate values. There are several functions to do this including **compute.real** but the most complete and easiest to use is **covariate.predictions**. Below we compute the value of survival for young in 1980 (**index=1**) for a range of values for weight and then plot the predicted values with confidence intervals as a function of weight. Because we used **covariate.results** (a **marklist** of models) the predictions are averaged over the models in the list and the estimates of precision include model uncertainty. See the help file for detailed information about **covariate.predictions**. Note that the names of the fields in the dataframe must match the names of covariates that you used in the model (e.g., weight).

```
> minmass=min(dipper$weight)
> maxmass=max(dipper$weight)
> mass.values=minmass+(0:30)*(maxmass-minmass)/30
> Phibymass=covariate.predictions(covariate.results,data=data.frame(weight=mass.values),indices=c(1))
# Plot predicted model averaged estimates by weight with pointwise confidence intervals
> plot(Phibymass$estimates$covdata, Phibymass$estimates$estimate,
          type="l",lwd=2,xlab="Mass(kg)",ylab="Survival",ylim=c(0,1))
> lines(Phibymass$estimates$covdata, Phibymass$estimates$lcl,lty=2)
> lines(Phibymass$estimates$covdata, Phibymass$estimates$ucl,lty=2)
```

Now let's consider time-varying individual covariates. **RMark** contains a pre-programmed time-varying covariate which is either age or TSM (time-since-marking) but it is handled via the parameter structure rather than with an individual covariate with the data. But it is a good example, because it illustrates that the value of time-varying covariates need to be known for each animal at each occasion regardless of whether it was caught or not at the occasion. Thus, the time-varying covariate cannot be one that requires capturing and handling of the animal. Beyond, age an obvious candidate for a time-varying individual covariate for the CJS model is a trap-dependence covariate. The idea here is that animals that were caught on a previous occasion are more likely to be caught on the next occasion. If e_i is the value of the capture history at occasion i, then it becomes the time varying covariate for modeling p_{i+1}. In a CJS model only $p_2, \ldots, p_k$ are estimated for a history with k occasions, so the time varying covariates are $e_1, \ldots, e_{k-1}$ for those parameters. Below with the dipper data we construct a sequence of covariates labeled $td1981, \ldots, td1986$ that contain the capture history entry for the years 1980 to 1985 for each dipper. They are labeled with the 1981 to 1986 suffix because those will be the times for the capture probabilities if we use **begin.time=1980**. Had we not included the assignment of **begin.time**, the times would default to begin at 1, and the variables would have to be named **td2**,. . . ,**td7** to be properly handled by the formula. First we start with a function that creates the trap dependence variable. It was written as a function because it is general and could be used elsewhere; although it would have to be changed if the time intervals between occasions were not 1.

```
> create.td=function(ch,varname="td",begin.time=1)
#
#  Arguments:
#    ch - capture history vector (0/1 values only)
#    varname - prefix for variable name
#    begin.time - time for first occasion
#
#  Value:
#    returns a datframe with trap-dependent variables
#            named varnamet+1,...,varnamet+nocc-1
#            where t is begin.time and nocc is the
#            number of occasions
#
{
# turn vector of capture history strings into a vector of characters
  char.vec=unlist(strsplit(ch,""))
# test to make sure they only contain 0 or 1
  if(!all(char.vec %in% c(0,1)))
    stop("Function only valid for CJS model without missing values")
  else
  {
#    get number of occasions (nocc) and change it into a matrix of numbers
     nocc=nchar(ch[1])
     tdmat=matrix(as.numeric(char.vec),ncol=nocc,byrow=TRUE)
   #   remove the last column which is not used
     tdmat=tdmat[,1:(nocc-1)]
#   turn it into a dataframe and assign the field (column) names
     tdmat=as.data.frame(tdmat)
     names(tdmat)=paste(varname,(begin.time+1):(begin.time+nocc-1),sep="")
     return(tdmat)
  }
}
```

Next we follow with the script that adds the variables to the dipper data and then uses the time-varying covariate in a few models. Note that you only use the prefix (e.g., **td**) in the formula and **RMark** adds the relevant suffix for the parameter.

```
> do.dipper.td=function()
{
# get data and add the td time-varying covariate, process the data
# and create the design data
data(dipper)
dipper=cbind(dipper,create.td(dipper$ch,begin.time=1980))
dipper.processed=process.data(dipper,begin.time=1980)
dipper.ddl=make.design.data(dipper.processed)
# create additional p field adult
dipper.ddl$p$adult=0
dipper.ddl$p$adult[dipper.ddl$p$Age > 1]=1
# define models for Phi
Phi.dot=list(formula=~1)
```

```
# define models for p
p.td=list(formula=~td)
p.td.adult=list(formula=~td:adult)
p.td.time=list(formula=~td:time)
p.time.plus.td=list(formula=~time+td)
# create model list
cml=create.model.list("CJS")
# run and return models
return(mark.wrapper(cml,data=dipper.processed,ddl=dipper.ddl))
}
> td.results=do.dipper.td()
```

Rather than focusing on the results, let's look at the design matrices for the models involving the time-varying covariate. In the first model (~**td**), we see that it added each covariate that matched the correct time dependent covariate that matched the parameter for 1981 to 1986 which we can see with the call to PIMS are indices 2 through 7.

```
> td.results[[1]]$design.matrix
                        Phi:(Intercept) p:(Intercept) p:td
Phi g1 c1980 a0 t1980 "  1"             "0"           "0"
  p g1 c1980 a1 t1981   "0"             "1"           "td1981"
  p g1 c1980 a2 t1982   "0"             "1"           "td1982"
  p g1 c1980 a3 t1983   "0"             "1"           "td1983"
  p g1 c1980 a4 t1984   "0"             "1"           "td1984"
  p g1 c1980 a5 t1985   "0"             "1"           "td1985"
  p g1 c1980 a6 t1986   "0"             "1"           "td1986"

> PIMS(td.results[[1]],"p")
group = Group 1
     1981 1982 1983 1984 1985 1986
1980    2    3    4    5    6    7
1981         3    4    5    6    7
1982              4    5    6    7
1983                   5    6    7
1984                        6    7
1985                             7
```

Now, if the experiment was one in which the animals were released we might not want to have a trap dependence for the first occasion after the initial release because it might not reflect trap dependence. We can limit the effect to occasions other than the first after release by interacting **td** with the **adult** design covariate (~**adult:td**). Note that parameter 2 does not have the trap-dependence effect and thus **td1981** is not used.

```
> td.results[[2]]$design.matrix
                        Phi:(Intercept) p:(Intercept) p:td:adult
Phi g1 c1980 a0 t1980   "1"             "0"           "0"
  p g1 c1980 a1 t1981   "0"             "1"           "0"
  p g1 c1980 a2 t1982   "0"             "1"           "td1982"
  p g1 c1980 a3 t1983   "0"             "1"           "td1983"
  p g1 c1980 a4 t1984   "0"             "1"           "td1984"
```

```
    p g1 c1980 a5 t1985   "0"              "1"            "td1985"
    p g1 c1980 a6 t1986   "0"              "1"            "td1986"

  > PIMS(td.results[[2]],"p")
  group = Group 1
       1981 1982 1983 1984 1985 1986
  1980    2    3    4    5    6    7
  1981         2    4    5    6    7
  1982              2    5    6    7
  1983                   2    6    7
  1984                        2    7
  1985                             2
```

Now, if you thought that the trap dependence effect might vary by time, you could interact **time** with **td(~time:td)**. Note that here the time effect is only for those caught on the previous occasion. Bit of a strange model without the main effect for time.

```
   > td.results[[3]]$design.matrix
Phi:(Intercept) p:(Intercept) p:td:time1981 p:td:time1982 p:td:time1983 p:td:time1984 p:td:time1985 p:td:time1986
 Phi g1 c1980 a0 t1980   "1"        "0"      "0"        "0"        "0"        "0"        "0"        "0"
   p g1 c1980 a1 t1981   "0"        "1"      "td1981"   "0"        "0"        "0"        "0"        "0"
   p g1 c1980 a2 t1982   "0"        "1"      "0"        "td1982"   "0"        "0" n       "0"        "0"
   p g1 c1980 a3 t1983   "0"        "1       "0"        "0"        "td1983"    "0"        "0"       "0"
   p g1 c1980 a4 t1984   "0" n       "1"      "0" n       "0"        "0"        "td1984"   "0"        "0"
   p g1 c1980 a5 t1985   "0" n       "1"      "0"        "0"        "0"        "0"        "td1985"   "0"
   p g1 c1980 a6 t1986   "0"        "1"      "0"        "0"        "0"        "0"        "0"        "td1986"
```

Finally, another model might be one with a time effect and an additive trap dependence effect (**~time+td**).

```
  > td.results[[4]]$design.matrix
                Phi:(Intercept) p:(Intercept) p:time1982 p:time1983 p:time1984 p:time1985 p:time1986 p:td
Phi g1 c1980 a0 t1980 "1"          "0"          "0"        "0"        "0"        "0"        "0"     "0"
p g1 c1980 a1 t1981   "0"          "1" n         "0"        "0"        "0"        "0"        "0"     "td1981"
p g1 c1980 a2 t1982   "0"          "1"          "1"        "0"        "0"        "0"        "0"     "td1982"
p g1 c1980 a3 t1983   "0"          "1"          "0"        "1"        "0"        "0"        "0"     "td1983"
p g1 c1980 a4 t1984   "0"          "1"          "0"        "0"        "1"        "0"        "0"     "td1984"
p g1 c1980 a5 t1985   "0"          "1"          "0"        "0"        "0"        "1"        "0"     "td1985"
p g1 c1980 a6 t1986   "0"          "1"          "0"        "0"        "0"        "0"        "1"     "td1986"

 > PIMS(td.results[[4]],"p",simplified=F)
 group = Group 1
       1981 1982 1983 1984 1985 1986
  1980   22   23   24   25   26   27
  1981        28   29   30   31   32
  1982             33   34   35   36
  1983                  37   38   39
  1984                       40   41
  1985                            42
```

When **RMark** runs **MARK** with an individual covariate model, it does not standardize the covariates (**MARK** does this on the fly) and **MARK** computes the real parameter estimates using the mean of the covariate value which may not be particularly useful. We'll again demonstrate the use of **covariate.predictions** to show how you can get the predicted values of p with **td=0** and 1 using this final model 4. This is a useful example to show how you limit predictions for covariates to specific parameters because in this case each covariate only applies to one parameter. To do so, the dataframe that is passed to the function should contain a field named **index** which is the parameter index for the

non-simplified PIM which is shown above. We want to compute a value of p for **td=0** and **td=1** for each time which can be specified with indices 22 through 27. The following 3 commands create the necessary dataframe as we can tell from the output:

```
> cov.df=data.frame(rbind(diag(rep(1,6)),diag(rep(0,6))))
> names(cov.df)=paste("td",1981:1986,sep="")
> cov.df$index=rep(22:27,2)
>  cov.df
   td1981 td1982 td1983 td1984 td1985 td1986 index
1       1      0      0      0      0      0    22
2       0      1      0      0      0      0    23
3       0      0      1      0      0      0    24
4       0      0      0      1      0      0    25
5       0      0      0      0      1      0    26
6       0      0      0      0      0      1    27
7       0      0      0      0      0      0    22
8       0      0      0      0      0      0    23
9       0      0      0      0      0      0    24
10      0      0      0      0      0      0    25
11      0      0      0      0      0      0    26
12      0      0      0      0      0      0    27
```

The following gets the predicted values and plots them for **td=1** and **td=0** as 2 different lines:

```
> p.est=covariate.predictions(td.results[[4]],data=cov.df)
> plot(1981:1986,p.est$estimates$estimate[1:6],type="b",ylim=c(0,1),xlab="Time",
    ylab="Capture probability",pch=1)
> lines(1981:1986,p.est$estimates$estimate[7:12],type="b",pch=2)
> legend(x=1984,y=.2,legend=c("td=1","td=0"),pch=1:2)
```

C.17. Multi-strata example

So far we have only used the **CJS** model in describing the **RMark** package. Now we switch to giving some examples with some of the other models supported by **RMark** (Table C.1). In general, there is little difference in using any of the models within **RMark** except for differences in the model parameters and some subtle differences due to the model structure. Each of the models in **RMark** comes with an example data set which shows a sample of analyses which often mimic the results in the sample **MARK** **.dbf** for that model.

We start off with the **Multistrata** model because it is a fairly useful model and it follows naturally from a discussion of time-varying covariates. The strata in the **Multistratum** model can be viewed as a time-varying factor variable for each animal except that the stratum (state) for each animal need not be known at each occasion. For the **Multistrata** model we use the **mstrata** data that corresponds to the **mssurv** example that accompanies **MARK**. For the **Multistrata** model there are 3 parameters: ψ (transition), S (survival) and p (capture). There are additional design data for these parameters to accommodate the strata. The strata labels are determined by the alphabetic characters used in the encounter history and need not be **A** to **C** like in this example. Below we show summaries for the design data for ψ and p (S is similar) for this example:

```
> data(mstrata)
> mstrata.processed=process.data(mstrata,model="Multistrata")
> mstrata.ddl=make.design.data(mstrata.processed)
> summary(mstrata.ddl$Psi)

group  cohort age    time   stratum tostratum     Cohort            Age              Time
1:36   1:18   0:18   1: 6   A:12    A:12      Min.   :0.0000   Min.   :0.0000   Min.   :0.000
       2:12   1:12   2:12   B:12    B:12      1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:1.000
       3: 6   2: 6   3:18   C:12    C:12      Median :0.5000   Median :0.5000   Median :1.500
                                              Mean   :0.6667   Mean   :0.6667   Mean   :1.333
                                              3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:2.000
                                              Max.   :2.0000   Max.   :2.0000   Max.   :2.000
                                                  A                B
                                               Min.   :0.0000   Min.   :0.0000
                                               1st Qu.:0.0000   1st Qu.:0.0000
                                               Median :0.0000   Median :0.0000
                                               Mean   :0.3333   Mean   :0.3333
                                               3rd Qu.:1.0000   3rd Qu.:1.0000
                                               Max.   :1.0000   Max.   :1.0000
      C               toA              toB              toC
Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.0000
Median :0.0000   Median :0.0000   Median :0.0000   Median :0.0000
Mean   :0.3333   Mean   :0.3333   Mean   :0.3333   Mean   :0.3333
3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:1.0000
Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000

  > summary(mstrata.ddl$p)

group  cohort age   time  stratum     Cohort            Age             Time             A
1:18   1:9    1:9   2:3   A:6     Min.   :0.0000   Min.   :1.000   Min.   :0.000   Min.   :0.0000
       2:6    2:6   3:6   B:6     1st Qu.:0.0000   1st Qu.:1.000   1st Qu.:1.000   1st Qu.:0.0000
       3:3    3:3   4:9   C:6     Median :0.5000   Median :1.500   Median :1.500   Median :0.0000
                                  Mean   :0.6667   Mean   :1.667   Mean   :1.333   Mean   :0.3333
                                  3rd Qu.:1.0000   3rd Qu.:2.000   3rd Qu.:2.000   3rd Qu.:1.0000
                                  Max.   :2.0000   Max.   :3.000   Max.   :2.000   Max.   :1.0000
                                       B                C
                               Min.   :0.0000   Min.   :0.0000
                               1st Qu.:0.0000   1st Qu.:0.0000
                               Median :0.0000   Median :0.0000
                               Mean   :0.3333   Mean   :0.3333
                               3rd Qu.:1.0000   3rd Qu.:1.0000
                               Max.   :1.0000   Max.   :1.0000
```

For all of the parameters, a **stratum** factor variable is included in the design data and a dummy variable (0/1) is included and named with the stratum label. For ψ parameters which describe transition from one stratum to another stratum, there are both **stratum** and **tostratum** factor and dummy variables.

Additional design data can be added with **merge_design.covariates** which can add data based on group and time variables. But if you want to add design data that is specific to particular strata then you'll need to write your own code. You can use the **R** function **merge** or if it is just one or two covariates you can use specific **R** statements that add the covariate as in the following example that adds a distance covariate to the **mstrata** example.

```
> run.mstrata=function()
{
# Process data
mstrata.processed=process.data(mstrata,model="Multistrata")
# Create default design data
mstrata.ddl=make.design.data(mstrata.processed) # Add distance covariate
mstrata.ddl$Psi$distance=0
mstrata.ddl$Psi$distance[mstrata.ddl$Psi$stratum=="A"&mstrata.ddl$Psi$tostratum=="B"]=12
mstrata.ddl$Psi$distance[mstrata.ddl$Psi$stratum=="A"&mstrata.ddl$Psi$tostratum=="C"]=5
mstrata.ddl$Psi$distance[mstrata.ddl$Psi$stratum=="B"&mstrata.ddl$Psi$tostratum=="C"]=2
mstrata.ddl$Psi$distance[mstrata.ddl$Psi$stratum=="B"&mstrata.ddl$Psi$tostratum=="A"]=12
mstrata.ddl$Psi$distance[mstrata.ddl$Psi$stratum=="C"&mstrata.ddl$Psi$tostratum=="A"]=5
mstrata.ddl$Psi$distance[mstrata.ddl$Psi$stratum=="C"&mstrata.ddl$Psi$tostratum=="B"]=2
# Create formula
Psi.distance=list(formula=~distance)
Psi.distance.time=list(formula=~distance+time)
p.stratum=list(formula=~stratum)
S.stratum=list(formula=~stratum)
model.list=create.model.list("Multistrata")
mstrata.results=mark.wrapper(model.list,data=mstrata.processed,ddl=mstrata.ddl)
return(mstrata.results)
}
> mstrata.results=run.mstrata()
> mstrata.results
```

The code that creates the models in the **MARK** example (**mssurv**) can be found by typing **?mstrata** in **RMark** or can be run by typing **example(mstrata)**. Constructing models for the **Multistrata** parameters is essentially the same as with the CJS model with the exception of ψ which is different due to its unique structure. For each stratum, there are transition parameters to the other strata and the probability of remaining in the stratum is computed by subtraction. Thus, for the **mstrata** example there is a transition from **A** to **B** and **A** to **C** and **A** to **A** is computed by subtraction. The same holds for the other strata. Thus, the **stratum** and **tostratum** factors are not fully crossed by default.

```
> table(mstrata.ddl$Psi[,c("stratum","tostratum")])

        tostratum
stratum   A B C
        A 0 6 6
        B 6 0 6
        C 6 6 0
```

Thus, to specify the interaction of **stratum** and **tostratum** to estimate each ψ parameter without restriction you would use **Psi.s=list(formula=~-1+stratum:tostratum)** and to fit the model with time varying transitions the model the ψ specification would be

```
> Psi.sxtime=list(formula=~-1+stratum:tostratum:time)
```

The transition that is computed by subtraction can be changed with the **subtract.stratum** argument of the **make.design.data** function. For this example the default call is equivalent to:

```
> mstrata.ddl=make.design.data(mstrata.processed,parameters=
              list(Psi=list(subtract.stratum=c("A","B","C"))))
```

but they can also be set such that the same stratum is computed by subtraction for all stratum:

```
> mstrata.ddl=make.design.data(mstrata.processed,parameters=
             list(Psi=list(subtract.stratum=c("B","B","B"))))
```

which does provide fully-crossed **stratum** and **tostratum** factors.

```
> table(mstrata.ddl$Psi[,c("stratum","tostratum")])

       tostratum
stratum A C
      A 6 6
      B 6 6
      C 6 6
```

But, that may not be the best reason for the choice of setting the **subtract.stratum**. Sometimes the choice may be decided based on model convergence. Some choices will yield better convergence if one or more of the ψ parameters is at a boundary.

In some cases, you may want to choose the **subtract.stratum** because you want to specify some ψ values to be set to zero. The easiest way to constrain specific ψ to zero is to delete the design data because that is the default value. However, the ψ that you want to set so zero cannot be computed by subtraction, so you need to set the **subtract.stratum** appropriately. For example, what if you wanted to set **PsiAA=PsiBB=PsiCC=0**? That could be done with the following code for the **mstrata** example:

```
> mstrata.ddl=make.design.data(mstrata.processed,parameters=
             list(Psi=list(subtract.stratum=c("B","A","A"))))
> mstrata.ddl$Psi=mstrata.ddl$Psi[!(mstrata.ddl$Psi$stratum==
             "A"&mstrata.ddl$Psi$tostratum=="A"),]
> mstrata.ddl$Psi=mstrata.ddl$Psi[!(mstrata.ddl$Psi$stratum==
             "B"&mstrata.ddl$Psi$tostratum=="B"),]
> mstrata.ddl$Psi=mstrata.ddl$Psi[!(mstrata.ddl$Psi$stratum==
             "C"&mstrata.ddl$Psi$tostratum=="C"),]
> mymodel=mark(mstrata.processed,mstrata.ddl)
> summary(mymodel,show.fixed=T)

Real Parameter Psi
 Stratum:A To:A
  1 2 3
1 0 0 0
2   0 0
3     0

 Stratum:A To:C
          1         2         3
1 0.5014851 0.5014851 0.5014851
2           0.5014851 0.5014851
3                     0.5014851

 Stratum:B To:B
  1 2 3
```

```
  1 0 0 0
  2   0 0
  3     0

   Stratum:B To:C
            1         2         3
  1 0.5014063 0.5014063 0.5014063
  2           0.5014063 0.5014063
  3                     0.5014063

   Stratum:C To:B
            1         2         3
  1 0.4999394 0.4999394 0.4999394
  2           0.4999394 0.4999394
  3                     0.4999394


   Stratum:C To:C
    1 2 3
  1 0 0 0
  2   0 0
  3     0
```

In other cases, the choice may be determined based on the ability to restrict equality for specific transitions. For example, if you had an example with 2 strata (**A** & **B**) and you wanted to set **PsiAB=PsiBB** you could do that by setting **subtract.stratum=c("A","A")** and fitting the **intercept(constant)** model for ψ. That gets more difficult with 3 or more strata. However, sometimes you can use design data to create constraints. For example, with the **mstrata** data, if you wanted to fit **PsiAB=PsiBA=PsiCA** and **PsiAC=PsiBC=PsiCC**, then you could use the following **subtract.stratum** and formula:

```
  > data(mstrata)
  > mstrata.processed=process.data(mstrata,model="Multistrata")
  > mstrata.ddl=make.design.data(mstrata.processed,parameters
            =list(Psi=list(subtract.stratum=c("A","B","B"))))
  > mark(mstrata.processed,mstrata.ddl,model.parameters=list(Psi=list(formula=~toC)))

<...>

  Real Parameter Psi
   Stratum:A To:B
            1         2         3
  1 0.2175964 0.2175964 0.2175964
  2           0.2175964 0.2175964
  3                     0.2175964

   Stratum:A To:C
            1         2         3
  1 0.2132953 0.2132953 0.2132953
  2           0.2132953 0.2132953
  3                     0.2132953
```

```
 Stratum:B To:A
          1         2         3
1 0.2175964 0.2175964 0.2175964
2           0.2175964 0.2175964
3                     0.2175964

 Stratum:B To:C
          1         2         3
1 0.2132953 0.2132953 0.2132953
2           0.2132953 0.2132953
3                     0.2132953

 Stratum:C To:A
          1         2         3
1 0.2175964 0.2175964 0.2175964
2           0.2175964 0.2175964
3                     0.2175964

 Stratum:C To:C
          1         2         3
1 0.2132953 0.2132953 0.2132953
2           0.2132953 0.2132953
3                     0.2132953
```

Now because the other parameters are computed by subtraction, it also set **PsiAA=PsiBB=PsiCB**.

What if you only wanted to set **PsiBC=PsiCC**? First, you could define a dummy variable **bc.toC** that was 1 for strata **B** and **C** for the transitions to **C** as follows:

```
> mstrata.ddl$Psi$bc.toC=0
> mstrata.ddl $Psi$bc.toC [mstrata.ddl $Psi$stratum%in%c("B","C")&
         mstrata.ddl $Psi$tostratum=="C"]=1
```

Then using the same **subtract.stratum** values you would naturally try:

```
mark(mstrata.processed,mstrata.ddl,model.parameters=list(Psi=list(formula=~bc.toC)))

 Real Parameter Psi
  Stratum:A To:B
          1        2        3
1 0.222929 0.222929 0.222929
2          0.222929 0.222929
3                   0.222929

  Stratum:A To:C
          1        2        3
1 0.222929 0.222929 0.222929
2          0.222929 0.222929
3                   0.222929
```

```
 Stratum:B To:A
          1         2         3
1 0.1731952 0.1731952 0.1731952
2           0.1731952 0.1731952
3                     0.1731952

 Stratum:B To:C
          1         2         3
1 0.3962874 0.3962874 0.3962874
2           0.3962874 0.3962874
3                     0.3962874

 Stratum:C To:A
          1         2         3
1 0.1731952 0.1731952 0.1731952
2           0.1731952 0.1731952
3                     0.1731952

 Stratum:C To:C
          1         2         3
1 0.3962874 0.3962874 0.3962874
2           0.3962874 0.3962874
3                     0.3962874
```

You might have been expecting that **PsiAB=PsiBA=PsiCA=PsiAC** but now we get **PsiAB=PsiAC** and **PsiCA=PsiBA** but the pairs differ. From the design matrix with just 2 columns you would not expect to get 3 different estimates. To understand what is happening you need to understand the **mlogit** link and how it relates to the β's. Below are the equations for each of the above ψ parameters using β_0 as the intercept and β_1 as the value for **bc.toC**:

$$\psi^{AB} = \psi^{AC} = \frac{\exp(\beta_0)}{1 + \exp(\beta_0) + \exp(\beta_0)}$$

$$\psi^{BA} = \psi^{CA} = \frac{\exp(\beta_0)}{1 + \exp(\beta_0) + \exp(\beta_0 + \beta_1)}$$

$$\psi^{BC} = \psi^{CC} = \frac{\exp(\beta_0 + \beta_1)}{1 + \exp(\beta_0) + \exp(\beta_0 + \beta_1)}$$

Due to the way the **mlogit** link is constructed, if you restrict parameters across a partial subset of the strata, then it may not be possible to construct the model you want. The solution is to change the link function to **logit** as shown below.

```
> mark(mstrata.processed,mstrata.ddl,model.parameters
         =list(Psi=list(formula=~bc.toC,link="logit")))

Real Parameter Psi
 Stratum:A To:B
          1         2         3
```

```
1 0.1993645 0.1993645 0.1993645
2           0.1993645 0.1993645
3                     0.1993645

 Stratum:A To:C
          1         2         3
1 0.1993645 0.1993645 0.1993645
2           0.1993645 0.1993645
3                     0.1993645

 Stratum:B To:A
          1         2         3
1 0.1993645 0.1993645 0.1993645
2           0.1993645 0.1993645
3                     0.1993645

 Stratum:B To:C
        1         2         3
1 0.3990723 0.3990723 0.3990723
2           0.3990723 0.3990723
3                     0.3990723

 Stratum:C To:A
          1         2         3
1 0.1993645 0.1993645 0.1993645
2           0.1993645 0.1993645
3                     0.1993645

 Stratum:C To:C
          1         2         3
1 0.3990723 0.3990723 0.3990723
2           0.3990723 0.3990723
3                     0.3990723
```

Why not use the **logit** link all of the time? You can do that but the **mlogit** link was chosen as the default for **RMark** because it provides a natural constraint to make sure the real values sum to 1. If you choose to use the **logit** link, then just beware that **MARK** enforces the constraint by penalizing the likelihood and that may not be as stable numerically. Clearly, to build some models you may be required to use the **logit** link. Make sure to look at the penalty value in the **MARK** output to make sure that the penalty value is 0. The **logit** link does have the additional advantage that the PIMS can be simplified whereas they cannot be simplified with the **mlogit** link. But, beware that some of the **RMark** code for computation on the results from **MARK** has been written specifically for the **mlogit** link.

The ψ estimates for the **subtract.stratum** are not given by **MARK**. Obviously, computing the point estimate is simple by summing the other values and subtracting from 1. However, computing the standard error and confidence interval is more tedious. To avoid doing this by hand, the function **TransitionMatrix** was created to compute each real parameter, standard error and confidence interval. See the help file for that function and **get.real** for more details. The following will run the example code for **mstrata** and then compute the transition matrix.

```
> example(mstrata)
> Psilist=get.real(mstrata.results[[1]],"Psi",vcv=T)
> Psivalues=Psilist$estimates

> TransitionMatrix(Psivalues[Psivalues$time==1,],vcv.real=Psilist$vcv.real)

$TransitionMat
          A         B         C
A 0.6020772 0.1993450 0.1985778
B 0.1993452 0.6020771 0.1985777
C 0.2003789 0.2003787 0.5992424

> $se.TransitionMat

           A          B          C
A 0.01863979 0.01412477 0.01413614
B 0.01412478 0.01863984 0.01413616
C 0.01422173 0.01422172 0.01871430

  > $lcl.TransitionMat

          A         B         C
A 0.5650384 0.1730952 0.1723155
B 0.1730954 0.5650382 0.1723153
C 0.1739486 0.1739485 0.5620711

> $ucl.TransitionMat

          A         B         C
A 0.6379827 0.2284757 0.2277414
B 0.2284760 0.6379827 0.2277414
C 0.2297083 0.2297081 0.6353057
```

C.18. Nest survival example

The nest survival model is quite different than most of the other models in **MARK** because it is **not** based on an encounter history. At present, neither **`convert.inp`** nor **`import.chdata`** will handle data entry for nest survival data. The data must be imported into an **R** dataframe and certain fields must be included with specific names. Two examples are provided in **RMark**. The killdeer example is the data that accompanies **MARK** and the mallard example provided by Jay Rotella is documented in

> Rotella, J. J., S. J. Dinsmore, T. L. Shaffer. 2004. Modeling nest-survival data: a comparison of recently developed methods that can be implemented in **MARK** and **SAS**. *Animal Biodiversity and Conservation* **27**:187-204.

The dataframe must contain the following variables with these names:

- **`FirstFound`**: day the nest was first found

- **LastPresent**: last day that a chick was present in the nest
- **LastChecked**: last day the nest was checked
- **Fate**: fate of the nest; 0=hatch and 1=depredated

It can also contain a field **Freq** which is the frequency of nests with this data. If it is always 1 then it is not needed. The dataframe can also contain any number of other covariate or identifier fields. If your dataframe contains a variable **AgeDay1**, which is the age of the nest on the first occasion then you can use a variable called **NestAge** in the formula which will create a set of time-dependent covariates named **NestAge1**,**NestAge2**,. . . ,**NestAge(nocc-1)** which will provide a way to incorporate the age of the nest in the model. The use of **AgeDay1** and **NestAge** was added because the age covariate in the design data for the parameter *S* (survival) assumes all nests are the same age and is not particularly useful. This effect could be incorporated by using the **add()** function in the design matrix but **RMark** does not have any capability for doing that and it is easier to create a time-dependent covariate to do the same thing.

The file **killdeer.inp** and **mallard.txt** come with **RMark**. The code below provides examples for importing and setting up nest survival data for **RMark**. Modify the path to **Rmark** as needed.

```
# EXAMPLE CODE FOR CONVERSION OF .INP TO NECESSARY DATA STRUCTURE
# read in killdeer.inp file
> killdeer=scan("C:/Program Files/R/R-2.6.0/library/RMark/data/killdeer.inp",
       what="character",sep="\n")
# strip out ; and write out all but first 2 lines which contain comments
> write(sub(";","",killdeer[3:20]),"killdeer.txt")
# read in as a dataframe from tab-delimited text file and assign names
> killdeer=read.table("killdeer.txt")
> names(killdeer)=c("id","FirstFound","LastPresent","LastChecked",
                              "Fate","Freq")
# Read in data, which are in a simple text file that
# looks like a MARK input file but (1) with no comments or semicolons and
# (2) with a 1st row that contains column labels
> mallard=read.table("C:/Program Files/R/R-2.6.0/library/RMark/data/mallard.txt",
  header=TRUE)
```

The help files for **killdeer** and **mallard** provide example code for analysis of nest survival data. In particular, the script in the **mallard** help file is a nice example constructed by Jay Rotella. It demonstrates the benefits of **RMark** and provides a useful model for scripting an entire analysis from model building to prediction and plotting. It uses an alternative approach with **find.covariates**, **fill.covariates** and **compute.real** functions which were created before **covariate.predictions**. We have extended this example further here to include a 3-D plot:

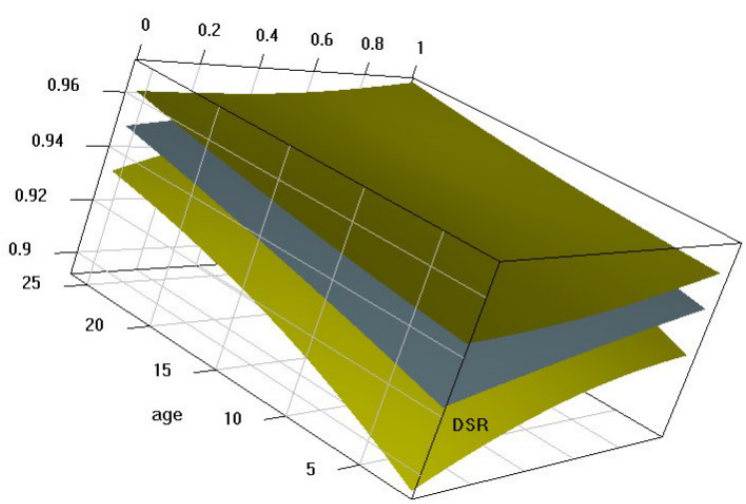
```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Example of use of RMark for modeling nest survival data - Mallard nests example        #
# The example runs the 9 models that are used in the Nest Survival chapter              #
# of the Gentle Introduction to MARK and that appear in Table 3 (page 198) of            #
# Rotella, J.J., S. J. Dinsmore, T.L. Shaffer.  2004.  Modeling nest-survival data:      #
#   a comparison of recently developed methods that can be implemented in MARK and SAS.  #
#   Animal Biodiversity and Conservation 27:187-204.                                     #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
> data(mallard)

# Treat dummy variables for habitat types as factors
> mallard$Native=factor(mallard$Native)
> mallard$Planted=factor(mallard$Planted)
```

```
> mallard$Wetland=factor(mallard$Wetland)
> mallard$Roadside=factor(mallard$Roadside)

# Examine a summary of the dataset
> summary(mallard)

# Write a function for evaluating a set of competing models
> run.mallard=function()
{
# 1. A model of constant daily survival rate (DSR)
Dot=mark(mallard,nocc=90,model="Nest",model.parameters=list(S=list(formula=~1)))

# 2. DSR varies by habitat type - treats habitats as factors
#  and the output provides S-hats for each habitat type
Hab=mark(mallard,nocc=90,model="Nest",
  model.parameters=list(S=list(formula=~Native+Planted+Wetland)),
  groups=c("Native","Planted","Wetland"))

# 3. DSR varies with vegetation thickness (Robel reading)
# Note: coefficients are estimated using the actual covariate
# values. They are not based on standardized covariate values.
Robel=mark(mallard,nocc=90,model="Nest",
  model.parameters=list(S=list(formula=~Robel)))

# 4. DSR varies with the amount of native vegetation in the surrounding area
# Note: coefficients are estimated using the actual covariate
# values. They are not based on standardized covariate values.
PpnGr=mark(mallard,nocc=90,model="Nest",model.parameters=list(S=list(formula=~PpnGrass)))

# 5. DSR follows a trend through time
TimeTrend=mark(mallard,nocc=90,model="Nest",model.parameters=list(S=list(formula=~Time)))

# 6. DSR varies with nest age
Age=mark(mallard,nocc=90,model="Nest",model.parameters=list(S=list(formula=~NestAge)))

# 7. DSR varies with nest age & habitat type
AgeHab=mark(mallard,nocc=90,model="Nest",
  model.parameters=list(S=list(formula=~NestAge+Native+Planted+Wetland)),
  groups=c("Native","Planted","Wetland"))

# 8. DSR varies with nest age & vegetation thickness
AgeRobel=mark(mallard,nocc=90,model="Nest",
  model.parameters=list(S=list(formula=~NestAge+Robel)))

# 9. DSR varies with nest age & amount of native vegetation in surrounding area
AgePpnGrass=mark(mallard,nocc=90,model="Nest",
  model.parameters=list(S=list(formula=~NestAge+PpnGrass)))

#
# Return model table and list of models
#
return(collect.models() )
}
```

```
> mallard.results=run.mallard()  # This runs the 9 models above and takes a minute or 2

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine table of model-selection results #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
> mallard.results                                      # print model-selection table to screen
> options(width=100)                                   # set page width to 100 characters
> sink("results.table.txt")                            # capture screen output to file
> print.marklist(mallard.results)                      # send output to file
> sink()                                               # return output to screen
> system("notepad results.table.txt",invisible=FALSE)    # view results in notepad

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine output for constant DSR model #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
> mallard.results$Dot                        # print MARK output to designated text editor
> mallard.results$Dot$results$beta           # view estimated beta for model in R
> mallard.results$Dot$results$real           # view estimated DSR estimate in R

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine output for 'DSR by habitat' model #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
> mallard.results$Hab                        # print MARK output to designated text editor
> mallard.results$Hab$design.matrix          # view the design matrix that was used
> mallard.results$Hab$results$beta           # view estimated beta for model in R
> mallard.results$Hab$results$beta.vcv       # view variance-covariance matrix for beta's
> mallard.results$Hab$results$real           # view the estimates of Daily Survival Rate

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine output for best model #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
> mallard.results$AgePpnGrass                # print MARK output to designated text editor
> mallard.results$AgePpnGrass$results$beta     # view estimated beta's in R
> mallard.results$AgePpnGrass$results$beta.vcv # view estimated var-cov matrix in R

# To obtain estimates of DSR for various values of 'NestAge' and 'PpnGrass'
#   some work additional work is needed.
# First, a simpler name for the object containing the preferred model results
> AgePpnGrass=mallard.results$AgePpnGrass
# Build design matrix with ages and ppn grass values of interest
> fc <- find.covariates(AgePpnGrass,mallard)
# iterate through sequence of ages and proportion grassland
#     values to build prediction surfaces
> seq.ages <- seq(2, 26, by=2)
> seq.ppn <- seq(0.01,0.99,length=89)
> point <- matrix(nrow=89, ncol=length(seq.ages))
> lower <- matrix(nrow=89, ncol=length(seq.ages))
> upper <- matrix(nrow=89, ncol=length(seq.ages))
> colnum <- 0
> for (iage in seq.ages) {
  fc$value[1:89]=iage                    # assign sequential age
  colnum <- colnum + 1
  fc$value[fc$var=="PpnGrass"]=seq.ppn # assign range of values to PpnGrass
  design=fill.covariates(AgePpnGrass,fc) # fill design matrix with values
```

```
  point[,colnum] <- compute.real(AgePpnGrass,design=design)[,"estimate"]
  lower[,colnum] <- compute.real(AgePpnGrass,design=design)[,"lcl"]
  upper[,colnum] <- compute.real(AgePpnGrass,design=design)[,"ucl"]
  }
#  Predicted surfaces shown in a window that can be rotated and zoomed by user
#     left mouse button=rotate, right mouse=zoom
> library(rgl)
> open3d()
> bg3d("white")
> material3d(col="black")
> persp3d(seq.ppn, seq.ages, point, aspect=c(1, 1, 0.5), col = "lightblue",
        xlab = "grass", ylab = "age", zlab = "DSR", zlim=range(c(upper,lower)),
        main="Daily survival rate (with CI)", sub="for model 'age and proportion grassland'")

> persp3d(seq.ppn, seq.ages, upper, aspect=c(1, 1, 0.5), col = "yellow", add=TRUE)

> persp3d(seq.ppn, seq.ages, lower, aspect=c(1, 1, 0.5), col = "yellow", add=TRUE)
> grid3d(c("x","y+","z"))
#  see rgl.snapshot(file="snapshot.png")  for creating png image of generated surfaces
```

The script fits 9 models to the data, then goes on to examine the best model and produce predicted point estimates and confidence intervals for a grid of values for the covariates (proportion native vegetation, and nest age). The 3 surfaces are then graphed, using a dynamic graphics library available in **R**, named **`rgl`**. This permits viewing of the surface by rotating and tilting, then capturing the most illuminating view of the surfaces (shown below).



C.19. Occupancy examples

The occupancy models are more similar to the encounter history models in **MARK** than the nest survival example but the histories relate to sites rather than animals and the values are presence(1)/absence(0) or counts of animals at a site.

At present there are 13 different occupancy models in **MARK** that are supported by **RMark**:

Occupancy, OccupHet, RDOccupEG, RDOccupPE, RDOccupPG, RDOccupHetEG, RDOccupHetPE, RDOccupHetPG, OccupRNPoisson, OccupRNNegBin, OccupRPoisson, OccupRNegBin, MSOccupancy.

Het means it uses the Pledger mixture for the detection probability model and those with RD are the robust design models. The 2 letter designations for the RD models are shorthand for the parameters that are estimated: 1) for EG, ψ, ϵ, and γ are estimated, 2) for PE, γ is dropped and 3) for PG, ϵ is dropped. For the latter 2 models, ψ can be estimated for each primary occasion.

The last 5 models include the Royle/Nichols count (**OccupRPoisson, OccupRNegBin**) and presence (**OccupRNPoisson, OccupRNNegBin**) models with Poisson and Negative Bionomial versions and the multi-state occupancy model (MSOccupancy). Each of the models and parameters are shown in Table C.1.

Example datasets are provided with **RMark** for each of the models. See the example datasets **salamander** and **weta** for Occupancy and OccupHet, **Donovan.7** for an example of OccupRNPoisson and OccupRNNegBin, **Donovan.8** for an example of OccupRPoisson and OccupRNegBin, **RDSalamander** for an example of the robust design models and **NicholsMSOccupancy** for an example of MSOccupancy. Here we provide more in-depth description and examples for the Occupancy and MSOccupancy models.

Imagine a scenario in which you wanted to model species-habitat dependent occupancy with detection probability dependent on effort which varied by occasion and site.

An example dataset (**mydata.txt**) might look as follows in a tab-delimited file with the variable names in the first row:

```
ch freq Species Habitat Effort1 Effort2 Effort3 Effort4 Effort5
00111 1 LGB Forest 5 2 14 2 5
00111 1 LGB Forest 5 3 16 3 5
10011 1 LGB Forest 4 2 11 2 4
10110 1 LGB Forest 5 3 15 3 5
00.00 1 LBB Forest 5 3 16 3 5
00000 1 LBB Forest 6 3 19 3 6
00010 1 LBB Forest 3 1 8 1 3
000.0 1 LBB Forest 5 3 16 3 5
00000 1 LBB Forest 4 2 13 2 4
00111 1 LBB Forest 4 2 12 2 4
00000 1 LBB Forest 5 2 14 2 5
00111 1 LBB Forest 3 2 10 2 3
00000 1 LBB Grassland 4 2 11 2 4
00000 1 LBB Grassland 3 2 10 2 3
00000 1 LBB Grassland 4 2 12 2 4
00000 1 LBB Grassland 2 1 6 1 2
00100 1 LGB Grassland 5 2 15 2 5
00100 1 LGB Grassland 4 2 13 2 4
```

Note the use of "." for cases in which as site was not visited. The file could be imported with the command

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

which denotes that **freq** is a numeric field (**"n"**), **Species** and **Habitat** are factor variables (**"f"**) and **Effort1** → **Effort5** are numeric. A **field.type** is not given for ch because it is always assumed to be a character string and *must always be the first field in each record*. The naming of **Effort1**→ **Effort5**

assumes that you'll use the default of **begin.time=1** because it is a time-varying covariate. An example run in **RMark** could be as follows:

```
# fit an additive Species+Habitat Psi model and Effort model for p
> mark(CovOccup,model="Occupancy",group=c("Species","Habitat"),
    model.parameters=list(Psi=list(formula=~Species+Habitat),p=list(formula=~Effort)))
```

Time-varying covariates are particularly useful for occupancy models because they relate to the site so they should always be known for each time (occasion). In most cases the time-varying covariates would be values like effort, weather, number of observers for detection probability but they could also be used for ψ. If the time-varying covariates are factor variables then you will need to create a dummy variable for the levels. For example, let's say you had 2 different observers doing the site visits and you thought that one observer might be more diligent than the other at searching. A factor variable with k levels requires $k-1$ dummy variables. In this case, $k = 2$, so we only need one dummy variable that we'll assign to **observer2**. If the site was visited by **observer2** on occasion j then the variable **observer2j** would be assigned a 1 and a 0 otherwise. You would need **observer21** → **observer2n** if you had n visits (occasions) to each site.

Some example data might look as follows in which observer 2 visited site 1 on occasions 2 and 4, and site 2 on occasions 1 and 5:

```
ch freq Species Habitat Observer21 Observer22 Observer23 Observer24 Observer25
00111 1 LGB Forest 0 1 0 1 0
00111 1 LGB Forest 1 0 0 0 1
```

The variable **Observer2** could be used in place of **Effort** in the example formulas shown above. If the factor covariate had more levels then you would have to add additional variables and they would also be included in the formula. You can think of those variables as you would columns in a design matrix except that their values would vary across sites/occasions. However, if you get many levels of the factor variable and many site visits, it is rather onerous to create all of those variables. Therefore, with a slightly clever usage of **model.matrix**, we created and added a function called **make.time.factor** to convert time-varying factor variables into a set of time-varying dummy variables. We demonstrate its usage with the **weta** dataset which is analyzed in MacKenzie *et al.* (2006) on pg 116-122, and which accompanies the program **PRESENCE** (which can be obtained from **http://www.mbr-pwrc.usgs.gov/software/presence.html**). From their Excel file we constructed the text file **weta.txt** which is in the data subdirectory of the **RMark** package. The following is the first few lines of the data:

```
ch        Browse        Obs1        Obs2        Obs3        Obs4        Obs5
0000.        1        1        3        2        3        .
0000.        1        1        3        2        3        .
0001.        1        1        3        2        3        .
0000.        0        1        3        2        3        .
0000.        1        1        3        2        3        .
```

The variables **Obs1** → **Obs5** contain the number of the observer that conducted the visit 1 to 5 and a "." if the site was not visited.

Each variable is read in as a factor variable with the following call to **import.chdata**:

```
> weta=import.chdata("weta.txt",field.types=c(rep("f",6)))
```

Each observer factor has 3 levels: 1,2,3 (excluding "."). To construct, dummy variables from a factor variable, one level is chosen as an intercept (observer 3 in this case) and you need $k-1$ (3-1=2) dummy variables for each time (visit). As with the time-varying effort variable above, these time-varying covariates have a suffix that creates a linkage with the time (visit). The following call to **make.time.factor**, creates those dummy variables (**Obs11**,. . . ,**Obs15**,**Obs21**,. . . ,**Obs25**) from $1 \rightarrow 5$ and replaces them in the data frame:

```
 > summary(weta)

      ch            Browse Obs1   Obs2   Obs3   Obs4   Obs5
 Length:72          0:37   .:19   .:15   .:12   .:24   .:28
 Class :character   1:35   1:21   1:17   1:20   1:18   1:15
 Mode  :character          2:12   2:20   2:20   2:15   2:14
                           3:20   3:20   3:20   3:15   3:15

> weta=make.time.factor(weta,"Obs",1:5,intercept=3)
> summary(weta)

      ch            Browse     Obs11            Obs21            Obs12            Obs22
 Length:72          0:37   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
 Class :character   1:35   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.0000
 Mode  :character          Median :0.0000   Median :0.0000   Median :0.0000   Median :0.0000
                           Mean   :0.2917   Mean   :0.1667   Mean   :0.2361   Mean   :0.2778
                           3rd Qu.:1.0000   3rd Qu.:0.0000   3rd Qu.:0.0000   3rd Qu.:1.0000
                           Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000
     Obs13            Obs23            Obs14          Obs24            Obs15
 Min.   :0.0000   Min.   :0.0000   Min.   :0.00   Min.   :0.0000   Min.   :0.0000
 1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.00   1st Qu.:0.0000   1st Qu.:0.0000
 Median :0.0000   Median :0.0000   Median :0.00   Median :0.0000   Median :0.0000
 Mean   :0.2778   Mean   :0.2778   Mean   :0.25   Mean   :0.2083   Mean   :0.2083
 3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:0.25   3rd Qu.:0.0000   3rd Qu.:0.0000
 Max.   :1.0000   Max.   :1.0000   Max.   :1.00   Max.   :1.0000   Max.   :1.0000
     Obs25
 Min.   :0.0000
 1st Qu.:0.0000
 Median :0.0000
 Mean   :0.1944
 3rd Qu.:0.0000
 Max.   :1.0000
```

Then the phrase **Obs1+Obs2** can be used in a formula to include an observer effect. See the help for **weta** or use **example(weta)** to explore the example further.

If by chance or design, a single observer was used for all sites on each occasion but the observers varied with occasion, then it isn't necessary to use a time-varying covariate and you can simply assign the observer level to the design data and use it in a model as follows (do not expect reasonable results with just 2 lines of data):

```
mydata.txt contents:
```

```
ch freq Species Habitat
00111 1 LGB Forest
00111 1 LGB Forest

 > CovOccup=import.chdata("mydata.txt",field.types=c("n",rep("f",2)))
 > CovOccup.process=process.data(CovOccup,model="Occupancy")
 > CovOccup.ddl=make.design.data(CovOccup.process)
 > CovOccup.ddl$p=merge_design.covariates(CovOccup.ddl$p,
      df=data.frame(time=1:5,observer=c("2","3","1","2","3")))

 > CovOccup.ddl$p
  group age time Age Time observer
1     1   0    1   0    0        2
2     1   1    2   1    1        3
3     1   2    3   2    2        1
4     1   3    4   3    3        2
5     1   4    5   4    4        3

 > mark(CovOccup.process,CovOccup.ddl,model.parameters=list(p=list(formula=~observer)))
```

For this simple case, we could have simply used an assignment statement to create observer, but had there been groups of sites, then **merge_design.covariates** is a better approach. Also, note the use of quote marks for the observer value to create a factor variable. If the quotes had not been used, the variable would have been improperly treated as a numeric variable (i.e., observer 2 effect is twice observer 1 effect).

Next, we'll move onto an example analysis of the **MSOccupancy** model that can be compared to the results in the manuscript by Nichols *et al*. (2007). We chose this model to demonstrate the relatively rare situation where parameters can share columns in the design matrix. As described in section C.3, most parameters do not share columns in the design matrix and for the exceptions, the argument **share=TRUE** or **FALSE** was added to the formula for the dominant parameter which was specified arbitrarily (p_1 in this case). In this case, p_1 and p_2 are detection probabilities for states 1 and 2 and often we will want to fit models where these parameters are equated or share covariate values. When **share=TRUE**, only a formula for the dominant parameter is specified but if **share=FALSE**, then a formula for both parameters are expected.

Nichols *et al*. (2007) specified 4 different models for detection probability: 1) variation in time but not by state (1/2), 2) time-invariant and $p_1 = p_2$, 3) time-invariant but $p1 \neq p_2$, and 4) time and state varying. For the first 2 models, p_1 and p_2 share columns in the design matrix and in the last 2 they do not share columns. The parameter specifications for these models are:

1. **p1=list(formula=~time,share=TRUE)**
2. **p1=list(formula=~1,share=TRUE)**
3. **p1=list(formula=~1,share=FALSE)**
   p2=list(formula=~1)
4. **p1=list(formula=~time,share=FALSE)**
   p2=list(formula=~time)

The script with these formulas that replicates the results of Nichols *et al*. (2007) is shown below. Note that there are some very minor differences in the AIC values which may be due to rounding.

```
# To create the data file use:
# NicholsMSOccupancy=convert.inp("NicholsMSOccupancy.inp")
#
# Create a function to fit the 12 models in Nichols et al (2007).
> do.MSOccupancy=function()
{
#  Get the data
   data(NicholsMSOccupancy)
# Define the models; default of Psi1=~1 and Psi2=~1 is assumed
# p varies by time but p1t=p2t
   p1.p2equal.by.time=list(formula=~time,share=TRUE)
# time-invariant p p1t=p2t=p1=p2
   p1.p2equal.dot=list(formula=~1,share=TRUE)
#time-invariant p1 not = p2
   p1.p2.different.dot=list(p1=list(formula=~1,share=FALSE),p2=list(formula=~1))
# time-varying p1t and p2t
   p1.p2.different.time=list(p1=list(formula=~time,share=FALSE),p2=list(formula=~time))
# delta2 model with one rate for times 1-2 and another for times 3-5; delta2 defined below
   Delta.delta2=list(formula=~delta2)
   Delta.dot=list(formula=~1)  # constant delta
   Delta.time=list(formula=~time) # time-varying delta
# Process the data for the MSOccupancy model
   NicholsMS.proc=process.data(NicholsMSOccupancy,model="MSOccupancy")
# Create the default design data
   NicholsMS.ddl=make.design.data(NicholsMS.proc)
# Add a field for the Delta design data called delta2.
# It is a factor variable with 2 levels: times 1-2, and times 3-5.
   NicholsMS.ddl=add.design.data(NicholsMS.proc,NicholsMS.ddl,
     "Delta",type="time",bins=c(0,2,5),name="delta2")
# Create a list using the 4 p models and 3 delta models (12 models total)
   cml=create.model.list("MSOccupancy")
# Fit each model in the list and return the results
   return(mark.wrapper(cml,data=NicholsMS.proc,ddl=NicholsMS.ddl))
}
# Call the function to fit the models and store it in MSOccupancy.results
> MSOccupancy.results=do.MSOccupancy()
# Print the model table for the results
> print(MSOccupancy.results)
# Adjust model selection by setting chat=1.74 used in the paper
> MSOccupancy.results=adjust.chat(chat=1.74,MSOccupancy.results)
# Print the adjusted model selection results table
> print(MSOccupancy.results)
```

The script also illustrates how to use **add.design.data** to accommodate their use of **delta2** and it also shows a feature that was recently added to **create.model.list** and **mark.wrapper**. These functions were originally designed to construct all possible combinations of parameter specifications. However, this example shows how those functions can now cope with lists that include more than one parameter specification. For example, the p_2 formula here will only be paired with the p_1 formula contained in the list

```
> p1.p2.different.time=list(p1=list(formula=~time,share=FALSE),p2=list(formula=~time))
```

and not with other p_1 formula where it would not be appropriate (i.e., **share=TRUE**).

C.20. Known fate example

The known fate model (**model="Known"**) has only one parameter S for survival. It uses the LD format for data entry. Below is the data description from the **MARK** help file (see also Chapters 2 and 16):

> *"The data coding for the known fate model requires a **1** in the **L** part of the encounter history for every occasion that the animal is alive at the start of the interval and its fate is known through the interval. A **10** means the animal lived through the interval, and a **11** means the animal died during the interval. There is no code of **01** allowed in the known fate model – this means that the animal was not alive at the start of the interval, so could not have died. To censor an animal for an interval where you don't know what was happening, use the **00** code. Thus, the encounter history **00101000101100** means that the animal lived through intervals 2 and 3, was censored for interval 4, lived through 5, and died in interval 6."*

We will use the **Blackduck** known-fate example that accompanies **MARK** and **RMark** and using **example(Blackduck)** to run some of the models that are in the **Blckduck.dbf/.fpt** files with **MARK**. Our main focus in this section will be to show some of the flexibility in **RMark** for handling time and age that can be useful with analysis of known fate data that span several years and ages with possibly overlapping time intervals. We will use (and likely abuse) the **Blackduck** example by arbitrarily dividing the data in half with the first half initiated 1 Jan 2000 and the second half in 1 Jan 2001. We'll also pretend that each occasion represents 0.25 years, so the 8 occasions represent 2 years. Thus, the data for 2000 will span 1 Jan 2000 - 31 Dec 2001 and the data for 2001 will span 1 Jan 2001 to 31 Dec 2002. Also, we'll have birds that were initially age 0 and age 1, so they can range in ages from 0 to 3 throughout the course of the data. The following code retrieves and defines the data with **year** and **BirdAge** changed to factor variables show they can be used to define groups:

```
data(Blackduck)
Blackduck$year=c(rep(2000,24),rep(2001,24))
Blackduck$year=factor(Blackduck$year)
Blackduck$BirdAge=factor(Blackduck$BirdAge)
```

Next we want to process the data and set the parameters to define the group, time and age structure that we have created. The group structure is defined with **groups=c("year","BirdAge")** and **age.var=2** specifies that the second group variable (**"BirdAge"**) should be treated as the factor for variable for defining initial ages. The age of the animals at the time of release for the 2 age groups were specified with **initial.age=c(0,1)**. They could have been different from the values of **BirdAge**. Next, the time intervals between occasions are set at 0.25 for each occasion. Finally, the initial time of release for each of the 4 groups is specified with **begin.time=c(2000,2001,2000,2001)**. Had the ordering of the group variables been swapped then it would have been specified as **begin.time=c(2000,2000,2001,2001)** with **age.var=1**.

```
> Blackduck.process=process.data(Blackduck,model="Known",groups=c("year","BirdAge"), age.var=2,
  initial.age=c(0,1), time.intervals=rep(.25,8), begin.time=c(2000,2001,2000,2001))
```

Now let's create and examine the design data to understand what has been done.

```
 > Blackduck.ddl=make.design.data(Blackduck.process)
 > Blackduck.ddl

$S
    group  age    time  Age Time year BirdAge
 1  20000 0.25 2000.25 0.25 0.00 2000       0
```

```
2  20000  0.5  2000.5 0.50 0.25 2000       0
3  20000 0.75 2000.75 0.75 0.50 2000       0
4  20000    1    2001 1.00 0.75 2000       0
5  20000 1.25 2001.25 1.25 1.00 2000       0
6  20000  1.5  2001.5 1.50 1.25 2000       0
7  20000 1.75 2001.75 1.75 1.50 2000       0
8  20000    2    2002 2.00 1.75 2000       0
9  20010 0.25 2001.25 0.25 1.00 2001       0
10 20010  0.5  2001.5 0.50 1.25 2001       0
11 20010 0.75 2001.75 0.75 1.50 2001       0
12 20010    1    2002 1.00 1.75 2001       0
13 20010 1.25 2002.25 1.25 2.00 2001       0
14 20010  1.5  2002.5 1.50 2.25 2001       0
15 20010 1.75 2002.75 1.75 2.50 2001       0
16 20010    2    2003 2.00 2.75 2001       0
17 20001 1.25 2000.25 1.25 0.00 2000       1
18 20001  1.5  2000.5 1.50 0.25 2000       1
19 20001 1.75 2000.75 1.75 0.50 2000       1
20 20001    2    2001 2.00 0.75 2000       1
21 20001 2.25 2001.25 2.25 1.00 2000       1
22 20001  2.5  2001.5 2.50 1.25 2000       1
23 20001 2.75 2001.75 2.75 1.50 2000       1
24 20001    3    2002 3.00 1.75 2000       1
25 20011 1.25 2001.25 1.25 1.00 2001       1
26 20011  1.5  2001.5 1.50 1.25 2001       1
27 20011 1.75 2001.75 1.75 1.50 2001       1
28 20011    2    2002 2.00 1.75 2001       1
29 20011 2.25 2002.25 2.25 2.00 2001       1
30 20011  2.5  2002.5 2.50 2.25 2001       1
31 20011 2.75 2002.75 2.75 2.50 2001       1
32 20011    3    2003 3.00 2.75 2001       1
```

As you can see, each of the 4 groups (year-age) has 8 S parameters and it assigns the proper time and age values; although it is labeling based on the end of the interval unlike with φ. This allows easy modeling of age and time effects even though the cohorts overlap and start at different times and ages. While it is possible to do the same with **MARK**, the pre-defined models are not setup correctly and the necessary bookkeeping with the PIMS is even more difficult because the time is not the same for each column in the PIM. In **RMark**, we can easily create **`Age`** and **`Time`** models as follows:

```
> mark(Blackduck.process,Blackduck.ddl,model.parameters=list(S=list(formula=~Time)))

<...>

Real Parameter S
                                1         2         3         4         5
Group:year2000.BirdAge0 0.4662898 0.5396722 0.6113742 0.6785598 0.7390873
Group:year2001.BirdAge0 0.7390873 0.7917159 0.8360831 0.8725213 0.9018106
Group:year2000.BirdAge1 0.4662898 0.5396722 0.6113742 0.6785598 0.7390873
Group:year2001.BirdAge1 0.7390873 0.7917159 0.8360831 0.8725213 0.9018106
                                6         7         8
```

```
Group:year2000.BirdAge0 0.7917159 0.8360831 0.8725213
Group:year2001.BirdAge0 0.9249493 0.9429801 0.9568809
Group:year2000.BirdAge1 0.7917159 0.8360831 0.8725213
Group:year2001.BirdAge1 0.9249493 0.9429801 0.9568809

 mark(Blackduck.process,Blackduck.ddl,model.parameters=list(S=list(formula=~Age)))

<...>

Real Parameter S
                                1         2         3         4         5
Group:year2000.BirdAge0 0.8116342 0.8024874 0.7930097 0.7832001 0.7730587
Group:year2001.BirdAge0 0.8116342 0.8024874 0.7930097 0.7832001 0.7730587
Group:year2000.BirdAge1 0.7730587 0.7625866 0.7517865 0.7406622 0.7292189
Group:year2001.BirdAge1 0.7730587 0.7625866 0.7517865 0.7406622 0.7292189
                                6         7         8
Group:year2000.BirdAge0 0.7625866 0.7517865 0.7406622
Group:year2001.BirdAge0 0.7625866 0.7517865 0.7406622
Group:year2000.BirdAge1 0.7174632 0.7054034 0.6930490
Group:year2001.BirdAge1 0.7174632 0.7054034 0.6930490

 mark(Blackduck.process,Blackduck.ddl,model.parameters=list(S=list(formula=~Age+Time)))

<...>

Real Parameter S
                                1         2         3         4         5
Group:year2000.BirdAge0 0.6781671 0.6936336 0.7086762 0.7232748 0.7374130
Group:year2001.BirdAge0 0.9380706 0.9421129 0.9459066 0.9494650 0.9528010
Group:year2000.BirdAge1 0.2809199 0.2956494 0.3108173 0.3264028 0.3423816
Group:year2001.BirdAge1 0.7374130 0.7510774 0.7642580 0.7769480 0.7891434
                                6         7         8
Group:year2000.BirdAge0 0.7510774 0.7642580 0.7769480
Group:year2001.BirdAge0 0.9559270 0.9588550 0.9615962
Group:year2000.BirdAge1 0.3587260 0.3754053 0.3923856
Group:year2001.BirdAge1 0.8008429 0.8120479 0.8227619
```

C.21. Exporting to MARK interface

Not all of the features in **MARK** are in `RMark` (e.g., median-$\hat{c}$), so it is useful to be able to export from **RMark** and import into the **MARK** interface. If you have read elsewhere (like in the previous version of C.21) about using the function **`export.chdata`** and **`export.model`** to export data and models into **MARK** interface, do **not** use that approach. Even though there was 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, the function **`export.MARK`** was implemented, and Gary White added the '**`File | RMark Import`**' menu item to the **MARK** interface. As an example, the following will export the ubiquitous dipper data file and the models contained in **`dipper.results`**

created in **example(dipper)**.

```
> example(dipper)
> dipper.processed=process.data(dipper,groups=("sex"))
> export.MARK(dipper.processed, "dipperproject", dipper.results)
```

If only NULL is returned then everything worked fine. The above code will create dipperproject.Rinp and dipperproject.inp and will rename all of the output files in **dipper.results** to have .tmp extensions so the **MARK** interface will know they need to be imported.

Next, open the **MARK** interface and choose '**File | RMark Import**' and browse to the location and select **dipperproject.Rinp**. **MARK** will create a .dbf and .fpt files with the names **dipperproject** and then populate with the models from **dipper.results**. Note that you must manually delete the .tmp files. If you let them remain in the directory and try to import from another project it will try to import those model results as well.

C.22. Using R for further computation and graphics

One of the nice side benefits of **RMark** is that all of the power of **R** is available for further computation and plotting with the **MARK** output which is brought into the **mark** model object. We recommend exploring the various online sources of help with **R** graphics.

In addition to graphics, the entire **R** environment is available for further computation with the results. Some of this has already been done for you with functions like **covariate.predictions** (C.16) and **TransitionMatrix** (C.17) but there are always different computations that can follow once an analysis is completed. The most important feature about the choice of **R** for a statistical environment and the reason for its expansive growth is that it is open source. There is literally a worldwide collection of programmers who are continually developing and adding code to the **R** environment. **R** as an open source environment has at least 4 important consequences for the **R** user:

1. cutting edge analysis techniques are always being added to the environment
2. anything you probably need for computation has already been written so search before you write new code
3. all of the source code (including **RMark**) is available for you to examine so you can learn from other **R** programmers, modify it for your own needs and you can see how the code works
4. as a user it is up to you to make sure you are using it properly and to verify that the results are accurate or at least make sense.

This last point applies equally to commercial software but the advantage with open source software is that you can look at the code. We'll finish this paragraph with one last bit of soapbox commentary about science and software. The **R** environment is a useful model for the scientific community because it is open source and thus transparent and available to anyone willing to learn.

Enough of that! So let's explore an example that frequently appears on the **MARK** support forum which is the computation of a Delta method variance. Appendix B provides a thorough explanation of the underlying theory, and how a Delta method variance can be constructed 'by hand'. While it is obviously important to understand how the calculations are done, and the general limits of the method, once you have done one or two by hand there is little gain in learning. So once the learning is done why not automate what you need? Not surprisingly there is **R** code available to construct Delta method

variances/covariances. As part of the **msm** package for analysis of multi-state Markov and hidden Markov processes, C. H. Jackson of the MRC Biostatistics Unit at Cambridge University provided a function called **deltamethod** for computation of Delta method variances of any function that can be differentiated with the **R** function **deriv** for symbolic differentiation of simple expressions. The code is quite simple because most of the work is in working out the derivatives and that is handled by the **deriv** function:

```
> deltamethod<- function (g, mean, cov, ses = TRUE)
{
    cov <- as.matrix(cov)
    n <- length(mean)
    if (!is.list(g))
        g <- list(g)
    if ((dim(cov)[1] != n) || (dim(cov)[2] != n))
        stop(paste("Covariances should be a ", n, " by ", n,
            " matrix"))
    syms <- paste("x", 1:n, sep = "")
    for (i in 1:n) assign(syms[i], mean[i])
    gdashmu <- t(sapply(g, function(form) {
        as.numeric(attr(eval(deriv(form, syms)), "gradient"))
    }))
    new.covar <- gdashmu %*% cov %*% t(gdashmu)
    if (ses) {
        new.se <- sqrt(diag(new.covar))
        new.se
    }
    else new.covar
}
```

If you install the **R** package **msm** (use **Packages/Install Packages**) from **CRAN**, then you can issue the command **library(msm)** to load the package and make the function **deltamethod** available in the **R** environment. In this example of using **deltamethod** we will compute the variances (or standard error, its square root) and covariances of φ and p from the **Phi(~1)p(~1)** model of the dipper data using the β's and their variances and covariances. This is not a particularly useful "further computation" but we chose it to show how **MARK** constructs these values and also to show that computation with the **deltamethod** function agrees with the **MARK** output. We start by fitting the model, displaying the summary with standard errors shown for the unique real parameters and extracting and displaying the β estimates:

```
 > data(dipper)
 > mymodel=mark(dipper,brief=TRUE)

Model: Phi(~1)p(~1)  npar= 2  lnl =  666.83766 AICc = 670.86603

 > summary(mymodel,se=TRUE,showall=FALSE)

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

Npar :  2
-2lnL:  666.8377
```

```
AICc :  670.866

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


Real Parameters
                 estimate        se       lcl       ucl fixed
Phi g1 c1 a0 t1 0.5602430 0.0251330 0.5105493 0.6087577
p g1 c1 a1 t2   0.9025835 0.0285857 0.8304826 0.9460113

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

We can see the confidence intervals for β's are simple normal 95% confidence intervals:

```
> beta.lcl=betas$estimate-1.96*betas$se
> beta.lcl
 [1] 0.04220351 1.58905157
> beta.ucl=betas$estimate+1.96*betas$se
> beta.ucl
 [1] 0.4420933 2.8634800
```

Now let's compute the confidence intervals for the real parameters which are the inverse **logit** (in general inverse of the chosen link function) of the lower and upper limits on the β's.

```
 > exp(beta.lcl)/(1+exp(beta.lcl))
     [1] 0.5105493 0.8304826
 > exp(beta.ucl)/(1+exp(beta.ucl))
     [1] 0.6087577 0.9460113
```

We can individually compute the standard errors for the real parameters with calls to **deltamethod** using the inverse **logit** function where **x1** refers to **beta**:

```
 > deltamethod(~exp(x1)/(1+exp(x1)),mean=betas$estimate[1],cov=betas$se[1]^2)
     [1] 0.02513295
 > deltamethod(~exp(x1)/(1+exp(x1)),mean=betas$estimate[2],cov=betas$se[2]^2)
     [1] 0.02858573
```

We can get the same results with a single call to **deltamethod** by using a list of functions and the variance-covariance matrix for **beta** which is in **results$beta.vcv**:

```
 > deltamethod(list(~exp(x1)/(1+exp(x1)),~exp(x2)/(1+exp(x2))),
              mean=betas$estimate,mymodel$results$beta.vcv)
      [1] 0.02513295 0.02858573
```

or we can get the variance-covariance matrix of the real parameters by setting **ses=FALSE**:

```
> deltamethod(list(~exp(x1)/(1+exp(x1)),~exp(x2)/(1+exp(x2))),
      mean=betas$estimate,mymodel$results$beta.vcv,ses=FALSE)
                [,1]          [,2]
  [1,]  0.0006316653 -0.0001842868
  [2,] -0.0001842868  0.0008171439
```

For closed population modeling, the **R** package **WiSP** (Wildlife Simulation Package, available from **http://www.ruwpa.st-and.ac.uk/estimating.abundance/WiSP/index.html**) can be used to simulate populations and sampling designs. Data simulated by **WiSP** can be converted into a form usable by **RMark** using a conversion function **transform.to.rmark()** found in the **WiSP** package. This enables the examination of estimator performance prior to the conduct of field experiments.

C.23. Problems and errors

The **RMark** code includes some error traps but there are a number of errors that can occur if the models or data are not setup properly. In no particular order, we give some errors that can occur, an explanation, and some possible fixes. We do not discuss **R** syntax errors which can occur easily if improper syntax is used.

As part of minimizing/checking erors, we suggest that you use an editor like **Tinn-R** (available from **https://sourceforge.net/projects/tinn-r**) that provides **R** syntax checking to develop scripts.

1. The following error message or one like it that occurs when **mark.exe** is running or afterwards, occurs when something is amiss with the data, model setup for **MARK** or you interrupted the job:

   ```
   Error in if (x4 > x2) { : argument is of length zero

   ********Following model failed to run : [name of model]**************
   S.dot.p.dot.Psi.dot
   Error in extract.mark.output(out, model, adjust) :
   MARK did not run properly. If error message was not shown, re-run
    MARK with invisible=FALSE
   ```

   Solution: Look at the most current input and output files in the directory to see if you can discern what happened. Error messages and the output will often move across the screen too quickly to read but you can always look at them with a text editor to discover the reason for the problem. The obtuse error above occurs because the output file is incomplete and the function **extract.mark.output** cannot find relevant fields in the output file.

2. The following error message occurs when a variable used in a formula cannot be found in the design data or as an individual covariate:

   ```
   Variable marked.as.adult used in formula is not defined in data
   Error in make.mark.model(data.proc, title = title, covariates = covariates,
   ```

   Solution: Check your the spelling of your variable name. Remember that **R** is case specific so check capitalization. If you added design data, make sure that you added the data to the design data for parameter that is generating

the error for this variable. If it is a time-varying covariate make sure that the times match the prefixes of the covariate names.

3. If you get an error like the following, you have created an incomplete factor variable in the design data.

```
Error in make.mark.model(data.proc, title = title, covariates = covariates,  :

Problem with design data. It appears that there are NA values
in one or more variables in design data for p
Make sure any binned factor completely spans range of data
```

   Solution: Examine the design data identified in the error message and redefine the factor variable.

4. If you get either of these error messages, there is a problem with the individual covariate that you have specified in the formula.

```
The following individual covariates are not allowed because
they are factor variables:
The following individual covariates are not allowed because
they contain NA:
```

   Solution: Use **`summary(data)`** where data is the dataframe containing your data. Examine the variable it names to see where it contains NA or if it is a factor variable. A factor variable cannot be used as an individual covariate. It is best to use factor variables in the group structure definition. If you want to use a factor variable as an individual covariate you need to create numeric dummy variables (0/1). See section C.16.

5. The following error occurs when you attempt to fix real parameters and the number of values does not match the number of indices.

```
Lengths of indices and values do not match for fixed parameters for p
```

   Solution: Refer to section C.11 to review how real parameters can be fixed at specific values.

6. The following error occurs when you specify a vector of initial values but the the number of values does not match the number of β's (number of columns in the design matrix).

```
Length of initial vector doesn't match design matrix
```

   Solution: Use another existing model to specify the initial values. or use **`model.matrix`** to compute the number of parameters will be fit. If the model has already run, extract the β estimates into a vector to verify the count or edit it and use as the initial values.

7. The following error occurs when you specify a formula that manages to create a design matrix which has all zeros for one or more real parameters (rows). This will most likely occur with the incorrect specification of interactions and the intercept is removed.

```
One or more formulae are invalid because the design matrix has all
zero rows for the following non-fixed parameters
```

Solution: Use **model.matrix** with the formula and design data and review the way you are constructing the formula.

8. The following error will occur if you pass the wrong data argument to **make.design.data**

```
Error in if (model == "CJS") par.list = c("Phi", "p") :
  argument is of length zero
```

Solution: Use the processed dataframe and not the original dataframe in the call the **make.design.data**.

It is reasonable to check an **RMark** model by creating it with the **MARK** interface and compare the results. If you do that, recognize that both interfaces use the same **mark.exe** to fit the model to the data. Thus, if there is a difference then it results from a difference in either the data or the model structure. Presumably, you are using the same data but it doesn't hurt to check the output from each model to see that it used the same data. A difference in the model is the most likely reason for any difference. If the deviance and the real parameters match but the β's are different, that is not a real concern because the same model can be fit with different beta structures. The differences in the β's can occur if different link functions are chosen or a different structure for the design matrix was used. However, if the deviances or real parameters are different then it should be investigated further. A difference in the link function can create difference in the real parameters if there is a problem with convergence of one of the models. However, the most likely difference is an error in the PIM structure or design matrix. Only a tedious search of the PIMS and design matrices will identify the problem. While it is always possible that there is an error in the **RMark** code, numerous examples have been tested in both pieces of software to check agreement.

C.24. A (very) brief R primer

There are a number of very good books and tutorials for learning **R**. See the **R** home page at http://www.r-project.org/ and browse the links under Documentation on the left panel. If you have questions, refer to the FAQ or use the Search utility. They are available from **R** Homepage or from within **R** under the Help menu. There is a phenomenal amount of material on the web that can help you get started with R. If you have problems and cannot find the answer with the materials on the web, a very active user group can be found on the R-help list server (see Mailing Lists on the home page). If you subscribe and post messages, please read the posting guide! The search utilities will search the **R**-help list server. There is a good chance your question has already been answered, so please read the FAQ and search before you post a question.

We have no intention of producing a full **R** tutorial here but we will provide some very beginner concepts and some others that are particularly relevant to using **RMark**. You can start **R** with the **R** icon or by double-clicking a **.Rdata** file which is an **R** workspace where everything is stored by **R**. If you start **R** with the icon, it will use the default **.Rdata** workspace located where you installed **R** (typically **c:/Program Files/R/Rvvvvv**, where **vvvvv** represents the version). You will most likely want to have more than one **.Rdata** workspace and there are many ways to create them, but the simplest is to use Windows to copy the default workspace and to paste it in whatever directory you choose. When you then double-click that particular **.Rdata** file, it will open it with **R**.

To avoid manually entering the command each time you initiate **R**, you can edit and enter the **library(RMark)** command into the file named "**RProfile.site**" with any text editor. It is located in the directory

```
C:\Program Files\R\R-v.v.v\etc\
```

where **v.v.v** represents the **R** version. If you add the library(RMark) command to rprofile.site, the RMark package will be loaded anytime you start R.The **RProfile.site** file is also a good place to make generic customizations to **R**. For example, adding the command **options(chmhelp=TRUE)** will mean that the help command will use the compiled help tool for windows. Or you can use options **(htmlhelp=TRUE)** to use the non-compiled html help. Either of those is better than using the default which does not allow hyperlinks. If you set up either help option you can enter "**?mark**" to see all the help categories for **RMark**. If you chose to use **htmlhelp**, click on index to see the complete list. The "**rprofile.site**" file is also a good place to change options like the default editor etc. See help for "options" and "Startup" from within **R**.

To avoid making changes to **RProfile.site** each time you update **R**, you can use the Windows ControlPanel and select System/Advanced/EnvironmentVariables to create an environment variable named **R_PROFILE**. For example, you can create a subdirectory,

```
C:\Program Files\R\RProfile\
```

and then copy your edited **Rprofile.site** to that subdirectory. Then define the environment variable **R_PROFILE** with the value

```
C:\Program Files\R\RProfile\RProfile.site
```

and it will always be used for each **R** session even if you update **R**.

You quit **R** with the command **q()** and it will ask whether you want to save the workspace image. If you select **No**, then any **R** objects you created/deleted/changed during the session will not be saved. You can save the workspace during the session with the File/Save Workspace or using the disk icon on the toolbar. This is not a bad idea to avoid losing work.

R is case-sensitive. **Q** and **q** are not the same. Some functions (e.g., **rowSums**) will even mix cases in the function name so be aware. Object names can have mixed cases, periods and underscore to improve readability (e.g., **my.list**, **squirrel_results**, **DipperModels**).

The symbol # is used for comments and anything to the right of the # is ignored in a line. You'll see them in examples below and in the help files for **RMark** and other help files.

The parentheses after **q** are necessary. Try typing **q** without the parentheses and what you'll get is a listing of the function "**q**". However when you type **q()**, it executes the function "**q**", and "**()**" represents the arguments for the function which happens to be empty in this case.

Almost everything you do in **R** will be executing functions that accept arguments and return a value. The functions and the values returned by executing functions can all be stored as named objects in the workspace. Assignment of function values to an object is done with the assignment operator which can be either **<-** or = (or **<<-** for global assignment within functions). Typing **x=1+1** or **x <- 1+1**, assigns the numeric result 2 to the object named x. Typing **x=mean(1:50)**, assigns the mean of the sequence of numbers from 1 to 50 to x. The definition of a function is an assignment of **R** code to an object with a specific function name. You can see a listing of the names of the objects in the workspace by typing **ls()**.

If you execute a function and do not assign the result (if any) to an object, a default **print** function for that object will display it on the screen or to a file (if you use the **sink** function) and nothing becomes of it. The function **ls()** is a good example. As it says in the help file for **ls**, the function returns a vector of character strings giving the names of the objects in the specified environment. That vector of character strings can be assigned to an object but if you simply type **ls()**, the character string is printed (displayed) on the screen.

Within **R** you can get help for any function by simply typing **?** followed by the function name. For example, **?ls**. If you don't know the name of the function but have some keywords to describe it, use the

function **help.search("my key words")** or browse through the help manuals which can be accessed from the Help menu. A reference card of commonly used **R** functions can also be useful - see

http://cran.r-project.org/doc/contrib/Short-refcard.pdf

You can access the full help file for **RMark** by opening the file **RMark.chm** (compiled help file) contained in the install directory **c:/Program Files/R/Rvvvvv/library/RMark/chm** (where **vvvvv** represents the **R** version number). If you only want to know the arguments of a function, you can use the function **args(function)** to list the argument names and default values of the function. For example, type **args(ls)** to see that the **ls** function does have arguments but the default values are typically used, so you only need to type **ls()**.

Values for function arguments can be assigned by their order in the function call and by specifying argument=value. The advantage of the latter format is that it is not order-specific. Both formats can be used in the same function call and it is a typical choice to specify the first few common arguments by order and then specifying less often used arguments by name. As an example, we will use the function **rnorm** which generates random values from a normal distribution. If you type **args(rnorm)**, you'll see the following: **function (n, mean = 0, sd = 1)**. This means that the function has 3 arguments named **n**, **mean** and **sd**. The latter 2 have default values of 0 and 1, so if you don't specify values for those arguments they will be assigned 0 and 1 respectively. The argument "**n**" is the number of random values to be generated and its value must be given because it has no default. By typing **rnorm(100)**, you will generate $n = 100$ random values from a standard normal distribution with mean 0 and standard deviation 1. If you don't assign them to an object they will simply be displayed on the screen. If you wanted an **sd** of 5, you could do that by either of the following calls: **rnorm(100,,5)** or **rnorm(100,sd=5)**. The first form assigns the argument values solely by position in the argument list and the second uses both formats with n being assigned 100 because of its first position and sd is assigned 5 with a named value. Naming arguments does make the code more readable and you could choose to specify all the arguments by name with **rnorm(n=100, mean=0, sd=5)** and that is perfectly suitable. If you forget to assign a value to an object which does not have a default, an error will be issued. For example, you may see something like the following:

```
> rnorm(mean=2)
 Error in rnorm(mean = 2) : argument ‘‘n’’ is missing, with no default
```

You can write your own functions that are stored in the workspace, but most functions you will use are in base **R** packages or other contributed packages that can be optionally installed with **R**. The multitude of contributed packages on CRAN (Comprehensive **R** Archive network) can be found from the **R** home page. While most contributed packages are on CRAN, there are other supported packages such as **RMark** that can be found on the web. Packages must be installed and then loaded with the **library()** function as described above for **RMark**. Once you have installed **RMark** and used the library(**RMark**) function you can access the help and use the functions. Remember that **R** is case-sensitive so **RMark** is not the same as **Rmark**.

There are several different kinds of data structures in **R**. The most basic is a vector and while most objects in **R** are generalizations of vectors, here we are referring to a vector as an ordered collection of items of the same type. For example, **c(4,2,1)** is a numeric vector with the first item being 4, the second 2 and the third being 1. As you might expect, **c()** is a function - the concatenate function that puts together items of the same type or coerces them to the same type. Numeric vectors can be created with sequences such as **1:5** which is the sequence from 1 to 5 and by various functions (e.g., **seq**, **rep**). Vectors can also contain character or logical values. For example, **c("apple","orange","grape")** is a vector of character strings of fruit names. Logical vectors are typically created by comparison operators

such as equality (==), not equal (!=), less than (<), greater than (>), not greater than (<=), and not less than (>=). Both of the following commands create a logical vector of length 5 with different values:

```
> 1:5 ==2
 [1] FALSE  TRUE FALSE FALSE FALSE
> 1:5 >2
 [1] FALSE FALSE  TRUE  TRUE  TRUE
Logical values (vector of length 1) can also be created with the %in% operator:
> "orange" %in% c("apple","orange","grape")
 [1] TRUE
> "pineapple" %in% c("apple","orange","grape")
 [1] FALSE
```

Other data structures include matrices, arrays, dataframes (tables) and lists. Matrices are rectangular structures with each element restricted to be the same type (e.g., numeric, character, logical) and arrays are generalizations of matrices to higher dimensions. While matrices are used in **RMark** (e.g., design matrix), the primary data structures are lists and dataframes. You need to know how to construct and manipulate both types of data structures to be able to run models other than the most rudimentary ones.

Lists are the most predominant data structure in **RMark** because they are used to specify argument values and most functions return lists. Dataframes are special cases of lists, so we'll start by describing lists. A list is a collection of data structures that are not restricted to be the same type/structure. A list can contain numeric vectors, character vectors, a matrix, other lists and so on and so forth. It is a truly generic structure that lets you paste together different kinds of data structures. A list is often the value returned from a function because a function can only return a single object and often it is the only way to return many different types of values by pasting them into a single list. Likewise, lists can be used to paste together different types of data (e.g, character, numeric, logical) that are related to be passed as value for a function argument.

Everyone is familiar with grocery and "to do" lists so we'll use them as examples. To create a very strange grocery list in **R** called groceries you would use the **`list()`** function:

```
> groceries=list(fruits=c("oranges","apples","grapes"),
            meat=c("steak", "chicken"), milk=c(1,.5) )
```

If you wanted to see the contents of groceries it would look as follows:

```
> groceries
 $fruits
 [1] "oranges" "apples"  "grapes"

 $meat
 [1] "steak"   "chicken"

 $milk
 [1] 1.0 0.5
```

The groceries list has length 3 as you can ascertain with the length function:

```
> length(groceries)
 [1] 3
```

It contains 3 vectors with the first being character vectors named "**fruits**" and "**meat**" and the third is a numeric vector containing the size in gallons and named "**milk**". The first vector contains 3 elements, and the second and third vectors each contain 2 elements. You can extract the **meat** vector in several ways.

```
> groceries$meat
 [1] "steak"   "chicken"
 >
> groceries[[2]]
 [1] "steak"   "chicken"
>
> groceries[["meat"]]
 [1] "steak"   "chicken"
```

The double-square brackets are used to extract a list element by number or name. What is returned is the contents of the list element which is a character vector in this case. Now notice what happens when you use single brackets instead:

```
> groceries[2]
 $meat
 [1] "steak"   "chicken"

> groceries["meat"]
 $meat
 [1] "steak"   "chicken"
```

Single brackets provide a subset of the list and the result is a list and not just the contents of the list. The difference between the single and double brackets is shown below with the is.list function. With single brackets the result is a list and with double brackets it is not a list.

```
> is.list(groceries["meat"])
 [1] TRUE
> is.list(groceries[["meat"]])
 [1] FALSE
```

In most cases with **RMark** you will use the [[]] to extract the contents of a single list element. On some occasions, you would like to extract several list elements and this is done with the single brackets:

```
> groceries[c("meat","milk")]
 $meat
 [1] "steak"   "chicken"

 $milk
 [1] 1.0 0.5

> groceries[2:3]
 $meat [
 1] "steak"   "chicken"

 $milk
 [1] 1.0 0.5
```

In both cases above, a list with 2 elements is returned. If you try to extract more than one list element with [[]], an error will result:

```
> groceries[[2:3]]
 Error in groceries[[2:3]] : subscript out of bounds

> groceries[[c("meat","milk")]]
 Error in groceries[[c("meat", "milk")]] : subscript out of bounds
```

In **RMark**, you'll most often extract specific list elements either by name (**`groceries$meat`**) or by position (**`groceries[[2]]`**).

Lists can contain lists as elements and while this can look rather bizarre at first, it is extremely handy. Assume that we had a "to do" list as follows:

```
todo=list(for.wife=c("grocery","cut grass","dry cleaner"),
for.me=c("watch tv","drink beer", "nap"))
```

If we concatenate the lists it merges them into a single list with all the different elements:

```
> c(todo,groceries)
 $for.wife
 [1] "grocery"     "cut grass"   "dry cleaner"

 $for.me
 [1] "watch tv"   "drink beer" "nap"

 $fruits
 [1] "oranges" "apples"  "grapes"

 $meat
 [1] "steak"   "chicken"

 $milk
 [1] 1.0 0.5
```

Alternatively, we can also create a list of lists as follows:

```
> mylists=list(todo=todo,groceries=groceries)
> mylists
 $todo
 $todo$for.wife
 [1] "grocery"     "cut grass"   "dry cleaner"

 $todo$for.me
 [1] "watch tv"   "drink beer" "nap"

 $groceries
 $groceries$fruits
 [1] "oranges" "apples"  "grapes"
```

```
$groceries$meat
[1] "steak"   "chicken"

$groceries$milk
[1] 1.0 0.5
```

I can extract each sub-list as described previously:

```
> mylists$todo
 $for.wife
 [1] "grocery"     "cut grass"   "dry cleaner"

 $for.me
 [1] "watch tv"   "drink beer" "nap"

> mylists$groceries
 $fruits
 [1] "oranges" "apples"  "grapes"

 $meat
 [1] "steak"   "chicken"

 $milk
 [1] 1.0 0.5
```

Lists of lists are used to provide a generic structure in **RMark** to accommodate varying parameters within the different **MARK** models. Likewise, design data for the parameters is represented as a list of dataframes.

That brings us to the final data structure we'll discuss. A dataframe is a specialized list in which each element is a named vector and each vector is of the same length but not necessarily of the same type. A dataframe is rectangular like a matrix. In a dataframe, all the values in a column (the list element vectors) are of the same type but one column can be numeric, the next column could be character and another could be logical. It is easiest to conceptualize dataframes as similar to tables like in ACCESS or any other database package.

We can show the link between lists and dataframes by using the **`as.data.frame`** function to convert the todo list to a dataframe.

```
> todo.data=as.data.frame(todo)
> todo.data
      for.wife     for.me
 1     grocery   watch tv 2   cut grass drink beer 3 dry cleaner nap
```

The columns of the dataframe **`todo.data`** are the 2 vectors contained in the **`todo`** list. This worked because each vector was of the same length. If we did the same conversion with the **`groceries`** list, an error occurs because the vectors are of unequal length:

```
> as.data.frame(groceries)
 Error in data.frame(fruits = c("oranges", "apples", "grapes"), meat
```

```
= c("steak",  :
        arguments imply differing number of rows: 3, 2
```

Notice that in the conversion from list to dataframe the quotation marks around the character strings vanished. Dataframes were designed for analysis and character strings aren't typically very useful for analysis but character strings can be used to represent the names of factor variable levels. By default, the character strings were coerced into a factor variable:

```
> todo.data$for.me
 [1] watch tv   drink beer nap Levels: drink beer nap watch tv
> is.factor(todo.data$for.me)
 [1] TRUE
```

The columns (variables) of **todo.data** are factor variables and the names of the levels for the factor variable **for.me** are "**drink beer**", "**nap**" and "**watch tv**". Note that the levels are alphabetized by default and the numeric values of **for.me** are determined by the order of the levels:

```
> as.numeric(todo.data$for.me)
[1] 3 1 2
```

In **RMark**, dataframes are used for the capture history and related covariate data for animals. They are also used for design data which describes the model structure.